**Prerequisites**

<https://docker-curriculum.com/>

<https://github.com/> 🡪 charleshoanduong1111 (UserID)

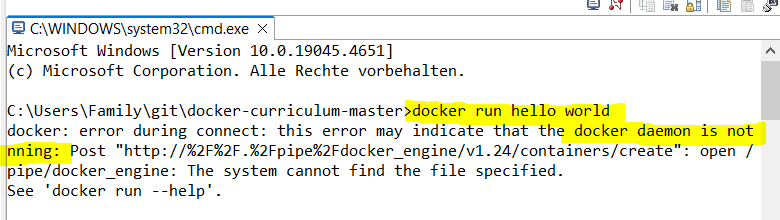
Docker Desktop installed 🡪 cduong1111 (UserID)

<https://hub.docker.com/> 🡪 charleshoanduong1111 (UserID)

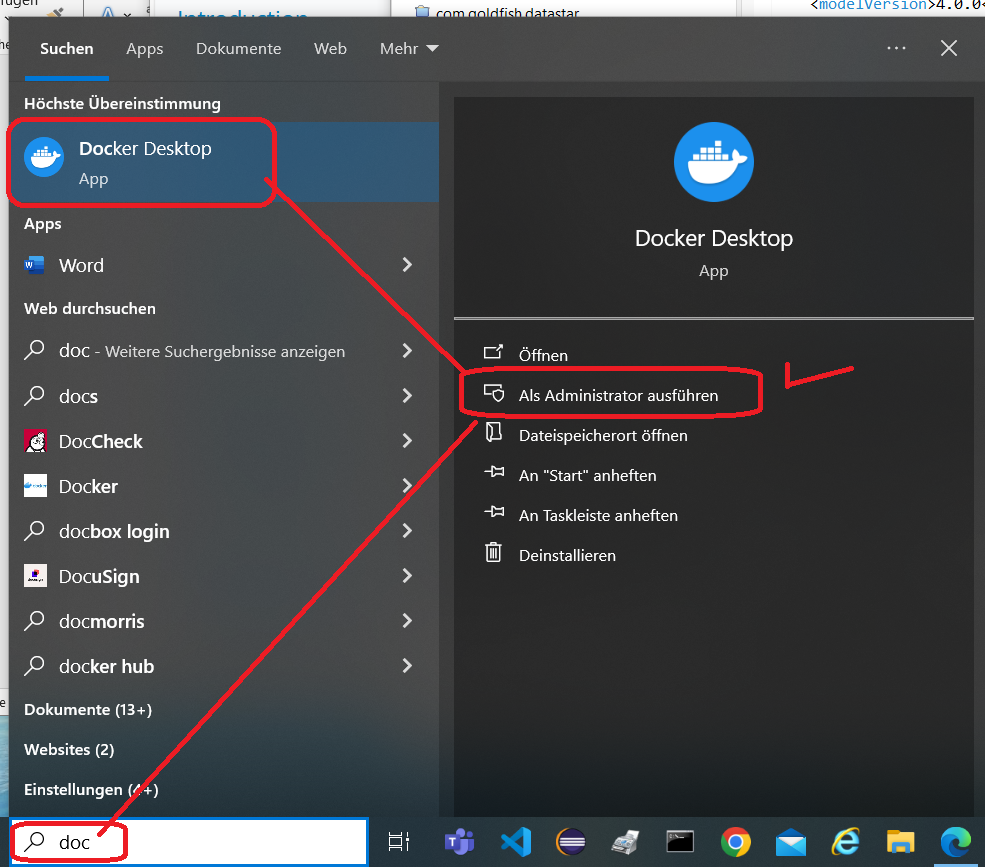
**Getting Started**

This document contains a series of several sections, each of which explains a particular aspect of Docker. In each section, we will be typing commands (or writing code). All the code used in the tutorial is available in the [Github repo](http://github.com/prakhar1989/docker-curriculum). 🡪 <https://github.com/prakhar1989/docker-curriculum>

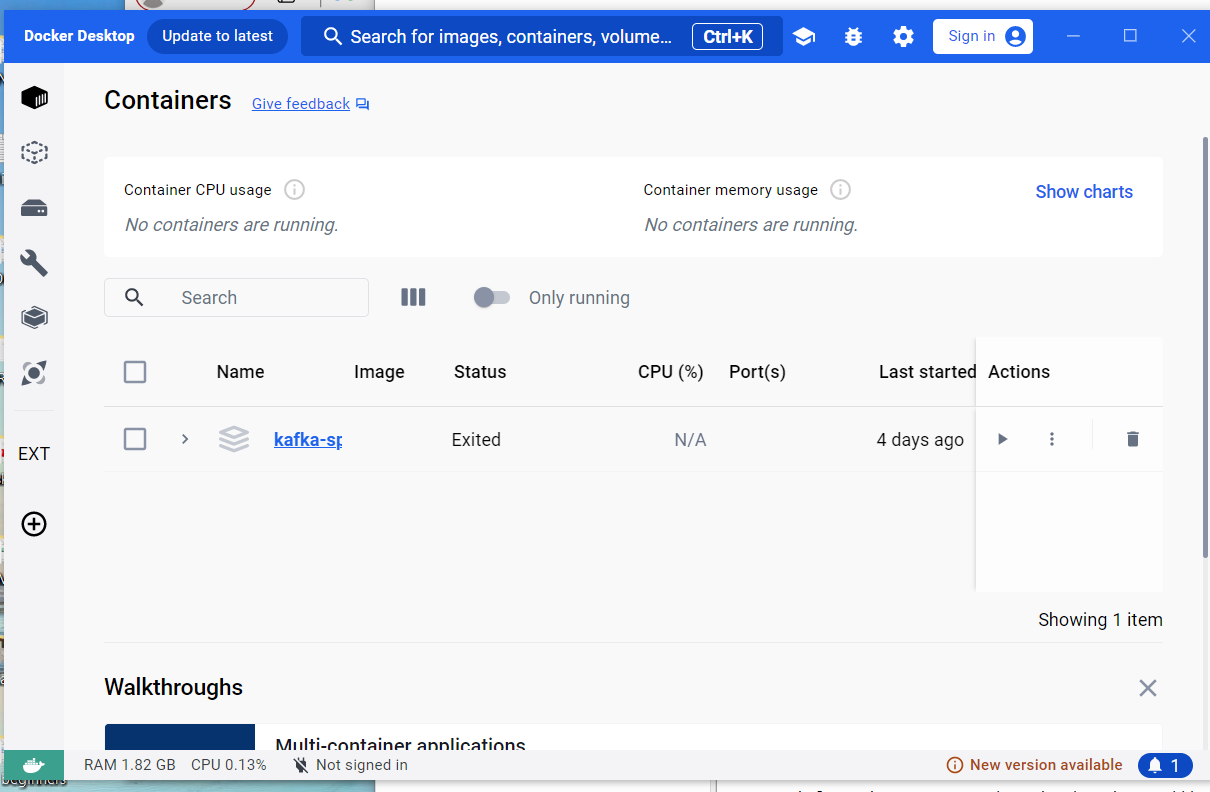
C:\Users\Family\git\docker-curriculum-master>docker run hello-world



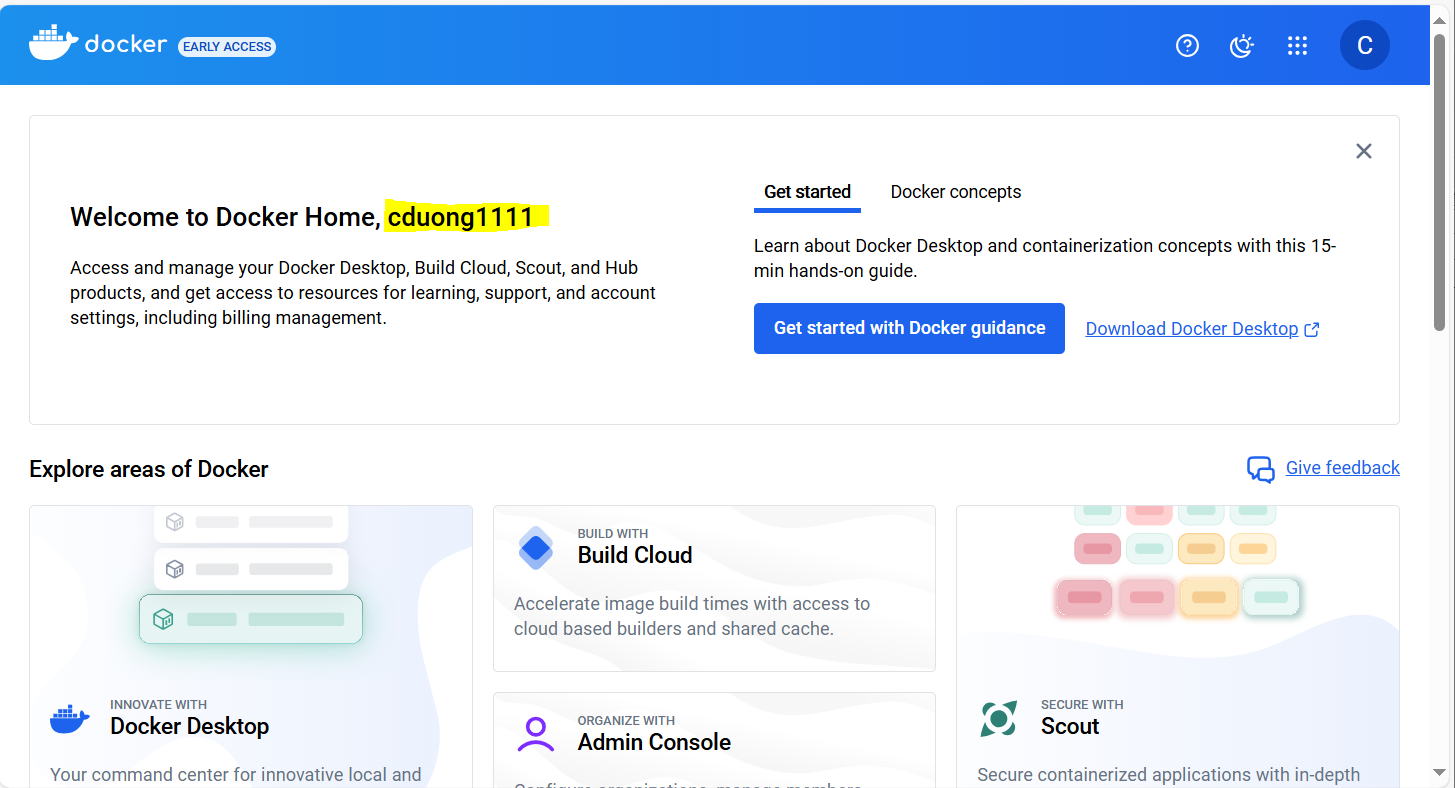
Open 🡪 Docker Desktop



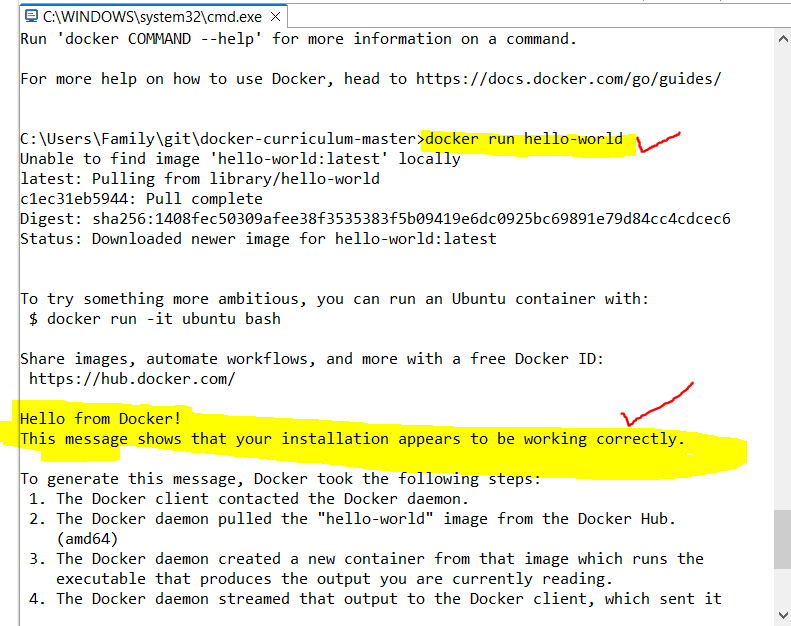
Docker Desktop opened



Login with userid cduong1111 instead of email.



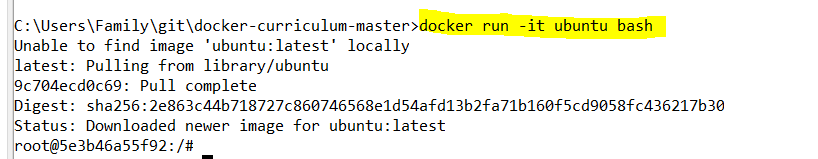
C:\Users\Family\git\docker-curriculum-master>docker run hello-world



To try something more ambitious, you can run an Ubuntu container with:

$ docker run -it ubuntu bash  
Share images, automate workflows, and more with a free Docker ID:  
 <https://hub.docker.com/>  
For more examples and ideas, visit:  
 <https://docs.docker.com/get-started/>

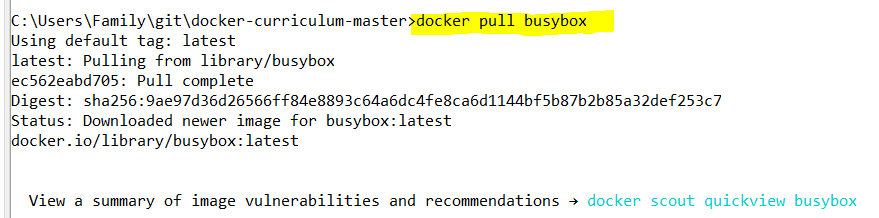
C:\Users\Family\git\docker-curriculum-master>docker run -it ubuntu bash



**Hello World**

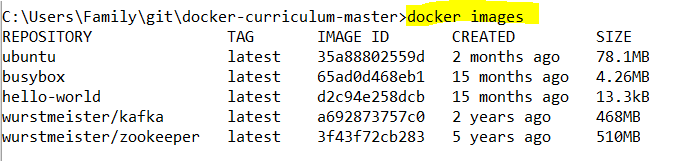
Playing with Busybox

C:\Users\Family\git\docker-curriculum-master>docker pull busybox



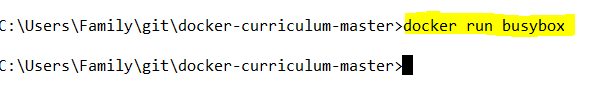
The pull command fetches the busybox [**image**](https://hub.docker.com/_/busybox/) from the [**Docker registry**](https://hub.docker.com/explore/) and saves it to our system. You can use the docker images command to see a list of all images on your system.

C:\Users\Family\git\docker-curriculum-master>docker images



Docker Run

Great! Let's now run a Docker **container** based on this image. To do that we are going to use the almighty docker run command.

C:\Users\Family\git\docker-curriculum-master>docker run busybox

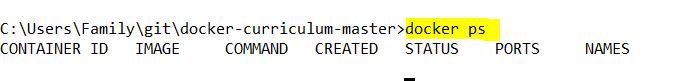
Wait, nothing happened! Is that a bug? Well, no. Behind the scenes, a lot of stuff happened. When you call run, the Docker client finds the image (busybox in this case), loads up the container and then runs a command in that container. When we run docker run busybox, we didn't provide a command, so the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

C:\Users\Family\git\docker-curriculum-master>docker run busybox echo "hello from busybox"



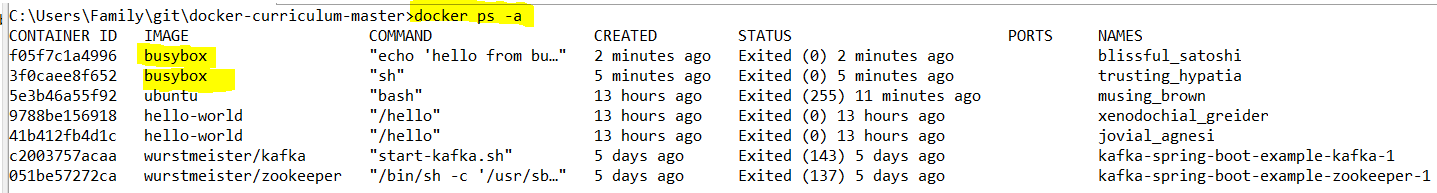
Nice - finally we see some output. In this case, the Docker client dutifully ran the echo command in our busybox container and then exited it. If you've noticed, all of that happened pretty quickly. Imagine booting up a virtual machine, running a command and then killing it. Now you know why they say containers are fast! Ok, now it's time to see the docker ps command. The docker ps command shows you all containers that are currently running.

C:\Users\Family\git\docker-curriculum-master>docker ps



Since no containers are running, we see a blank line. Let's try a more useful variant: docker ps -a

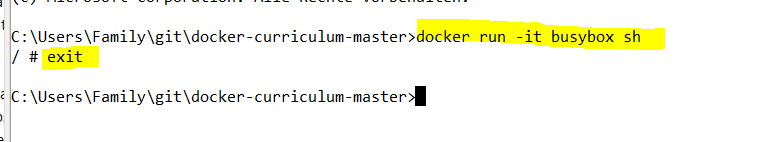
C:\Users\Family\git\docker-curriculum-master>docker ps -a



So what we see above is a list of all containers that we ran. Do notice that the STATUS column shows that these containers exited a few minutes ago.

You're probably wondering if there is a way to run more than just one command in a container. Let's try that now:

C:\Users\Family\git\docker-curriculum-master>docker run -it busybox sh



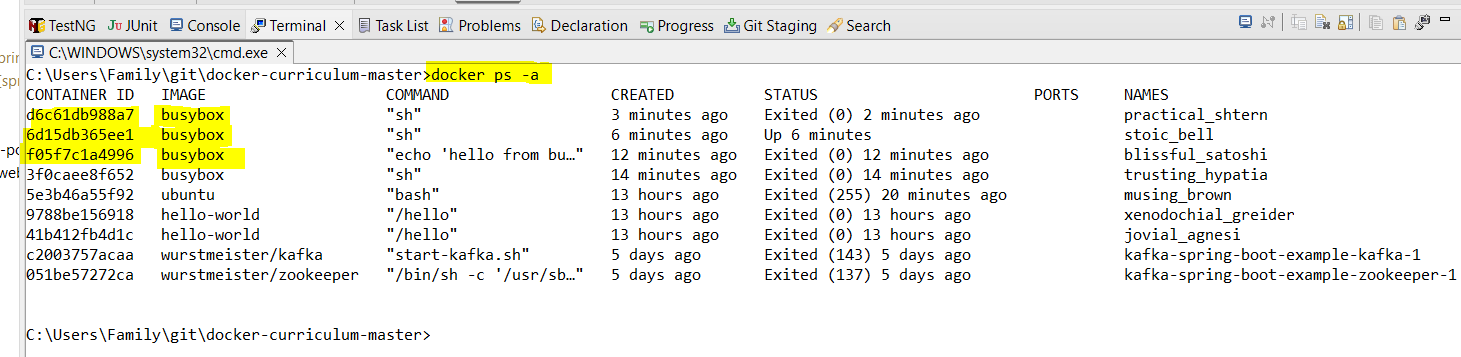
Running the run command with the -it flags attaches us to an interactive tty in the container. Now we can run as many commands in the container as we want. Take some time to run your favorite commands.

***Danger Zone****: If you're feeling particularly adventurous you can try rm -rf bin in the container. Make sure you run this command in the container and****not****in your laptop/desktop. Doing this will make any other commands like ls, uptime not work. Once everything stops working, you can exit the container (type exit and press Enter) and then start it up again with the docker run -it busybox sh command. Since Docker creates a new container every time, everything should start working again.*

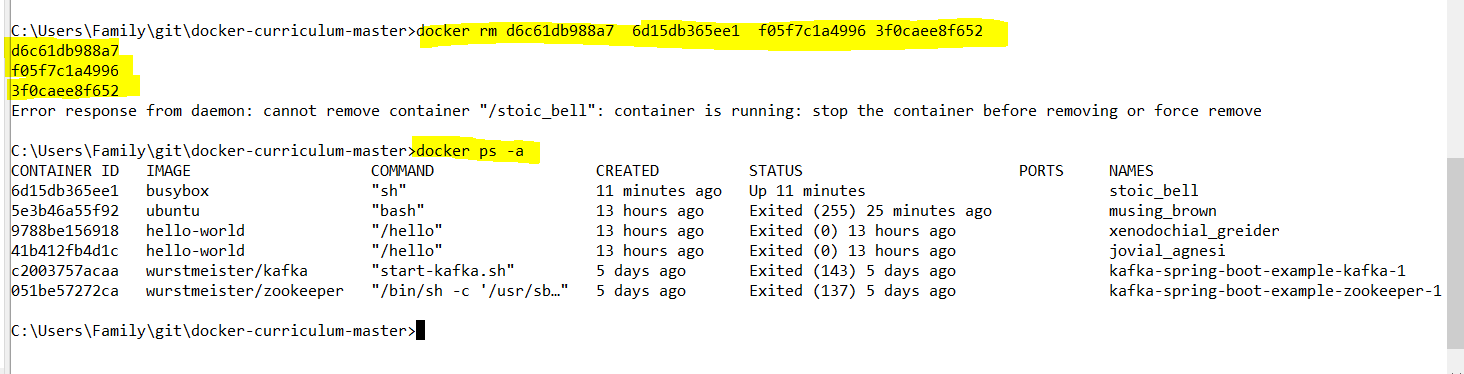
That concludes a whirlwind tour of the mighty docker run command, which would most likely be the command you'll use most often. It makes sense to spend some time getting comfortable with it. To find out more about run, use docker run --help to see a list of all flags it supports. As we proceed further, we'll see a few more variants of docker run.

Before we move ahead though, let's quickly talk about deleting containers. We saw above that we can still see remnants of the container even after we've exited by running docker ps -a. Throughout this tutorial, you'll run docker run multiple times and leaving stray containers will eat up disk space. Hence, as a rule of thumb, I clean up containers once I'm done with them. To do that, you can run the docker rm command. Just copy the container IDs from above and paste them alongside the command.

C:\Users\Family\git\docker-curriculum-master>docker ps -a



C:\Users\Family\git\docker-curriculum-master>docker rm d6c61db988a7 6d15db365ee1 f05f7c1a4996 3f0caee8f652



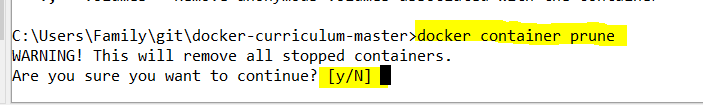
On deletion, you should see the IDs echoed back to you. If you have a bunch of containers to delete in one go, copy-pasting IDs can be tedious. In that case, you can simply run -

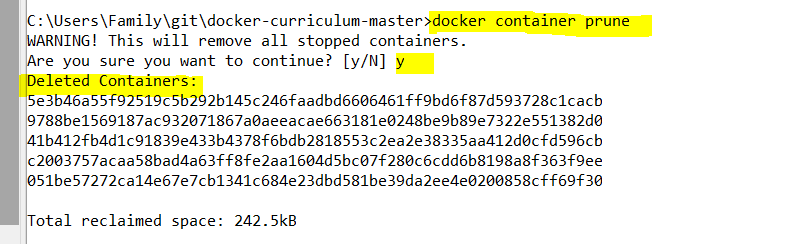
$ docker rm $(docker ps -a -q -f status=exited)

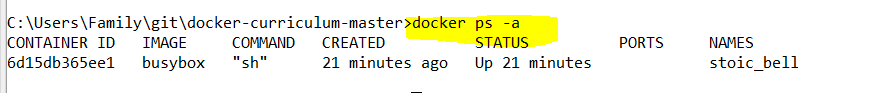
This command deletes all containers that have a status of exited. In case you're wondering, the -q flag, only returns the numeric IDs and -f filters output based on conditions provided. One last thing that'll be useful is the --rm flag that can be passed to docker run which automatically deletes the container once it's exited from. For one off docker runs, --rm flag is very useful.

In later versions of Docker, the docker container prune command can be used to achieve the same effect.

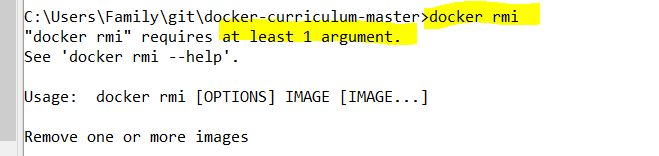
C:\Users\Family\git\docker-curriculum-master>docker container prune







Lastly, you can also delete images that you no longer need by running docker rmi.



Terminology

In the last section, we used a lot of Docker-specific jargon which might be confusing to some. So before we go further, let me clarify some terminology that is used frequently in 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 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](https://kitematic.com/) which provide a GUI to the users.
* *Docker Hub* - A [registry](https://hub.docker.com/explore/) 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.

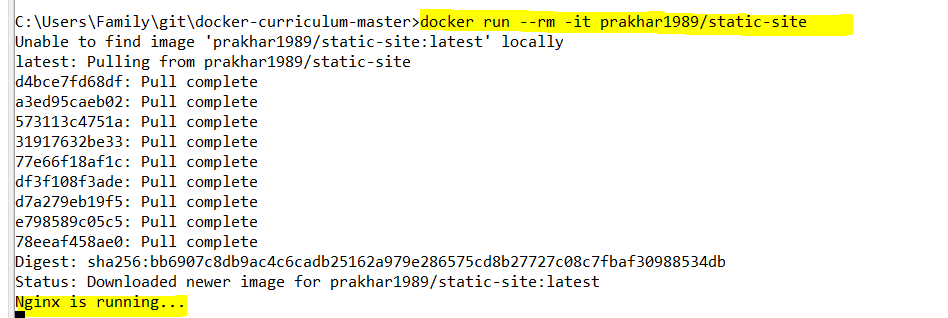
**Webapps with Docker**

Great! So we have now looked at docker run, played with a Docker container and also got a hang of some terminology. Armed with all this knowledge, we are now ready to get to the real-stuff, i.e. deploying web applications with Docker!

Static Sites

Let's start by taking baby-steps. The first thing we're going to look at is how we can run a dead-simple static website. We're going to pull a Docker image from Docker Hub, run the container and see how easy it is to run a webserver.

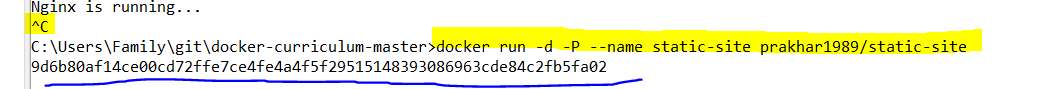
Let's begin. The image that we are going to use is a single-page [website](http://github.com/prakhar1989/docker-curriculum) that I've already created for the purpose of this demo and hosted on the [registry](https://hub.docker.com/r/prakhar1989/static-site/) - prakhar1989/static-site (<https://github.com/prakhar1989/docker-curriculum>). We can download and run the image directly in one go using docker run. As noted above, the --rm flag automatically removes the container when it exits and the -it flag specifies an interactive terminal which makes it easier to kill the container with Ctrl+C (on windows).

C:\Users\Family\git\docker-curriculum-master>docker run --rm -it prakhar1989/static-site 

Since the image doesn't exist locally, the client will first fetch the image from the registry and then run the image. If all goes well, you should see a Nginx is running... message in your terminal. Okay now that the server is running, how to see the website? What port is it running on? And more importantly, how do we access the container directly from our host machine? Hit Ctrl+C to stop the container.

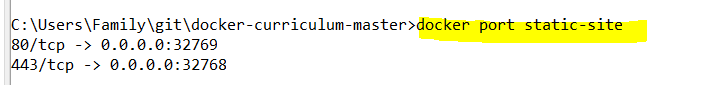
Well, in this case, the client is not exposing any ports so we need to re-run the docker run command to publish ports. While we're at it, we should also find a way so that our terminal is not attached to the running container. This way, you can happily close your terminal and keep the container running. This is called **detached** mode.

C:\Users\Family\git\docker-curriculum-master>docker run -d -P --name static-site prakhar1989/static-site

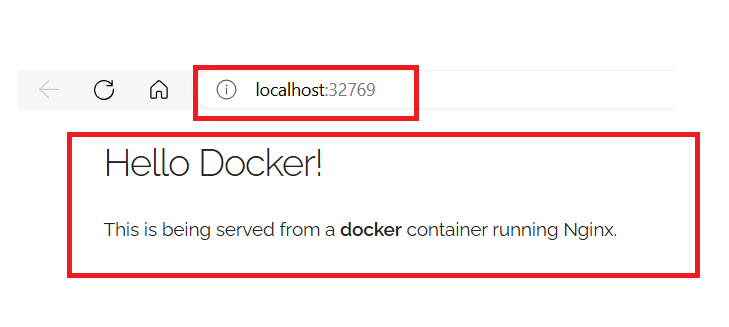


In the above command, -d will detach our terminal, -P will publish all exposed ports to random ports and finally --name corresponds to a name we want to give. Now we can see the ports by running the docker port [CONTAINER] command

C:\Users\Family\git\docker-curriculum-master>docker port static-site



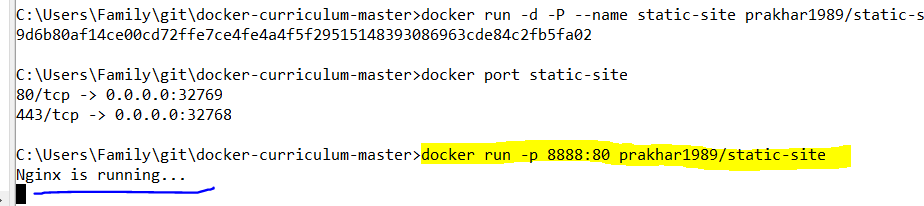
You can open [http://localhost:32769](http://localhost:32769/) in your browser.



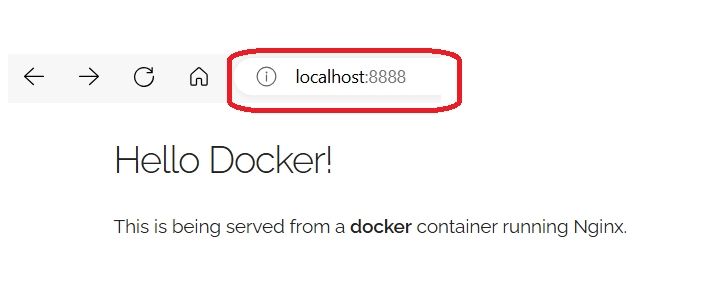
*Note: If you're using docker-toolbox, then you might need to use docker-machine ip default to get the IP.*

You can also specify a custom port to which the client will forward connections to the container.

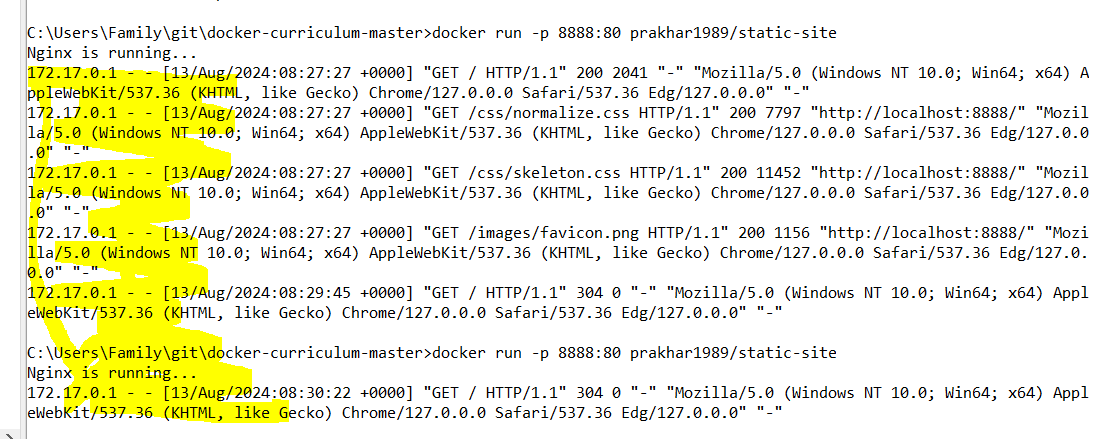
C:\Users\Family\git\docker-curriculum-master>docker run -p 8888:80 prakhar1989/static-site



You can open <http://localhost:8888> in your browser.

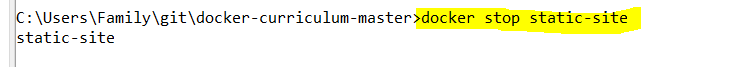


And you can see more information under Nginx is running….



To stop a detached container, run docker stop by giving the container ID. In this case, we can use the name static-site we used to start the container.

C:\Users\Family\git\docker-curriculum-master>docker stop static-site



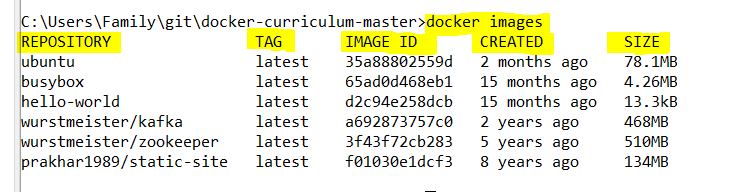
I'm sure you agree that was super simple. To deploy this on a real server you would just need to install Docker, and run the above Docker command. Now that you've seen how to run a webserver inside a Docker image, you must be wondering - how do I create my own Docker image? This is the question we'll be exploring in the next section.

Docker Images

We've looked at images before, but in this section we'll dive deeper into what Docker images are and build our own image! Lastly, we'll also use that image to run our application locally and finally deploy on [AWS](http://aws.amazon.com/) to share it with our friends! Excited? Great! Let's get started.

Docker images are the basis of containers. In the previous example, we **pulled** the *Busybox* image from the registry and asked the Docker client to run a container **based** on that image. To see the list of images that are available locally, use the docker images command.

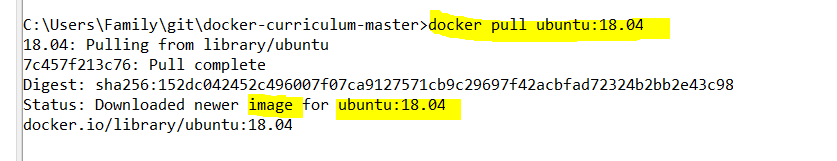
C:\Users\Family\git\docker-curriculum-master>docker images



The above gives a list of images that I've pulled from the registry, along with ones that I've created myself (we'll shortly see how). The TAG refers to a particular snapshot of the image and the IMAGE ID is the corresponding unique identifier for that image.

For simplicity, you can think of an image akin to a git repository - images can be [committed](https://docs.docker.com/engine/reference/commandline/commit/) with changes and have multiple versions. If you don't provide a specific version number, the client defaults to latest. For example, you can pull a specific version of ubuntu image

C:\Users\Family\git\docker-curriculum-master>docker pull ubuntu:18.04



To get a new Docker image you can either get it from a registry (such as the Docker Hub) or create your own. There are tens of thousands of images available on [Docker Hub](https://hub.docker.com/explore/). You can also search for images directly from the command line using docker search.

An important distinction to be aware of when it comes to images is the difference between base and child images.

* **Base images** are images that have no parent image, usually images with an OS like ubuntu, busybox or debian.
* **Child images** are images that build on base images and add additional functionality.

Then there are official and user images, which can be both base and child images.

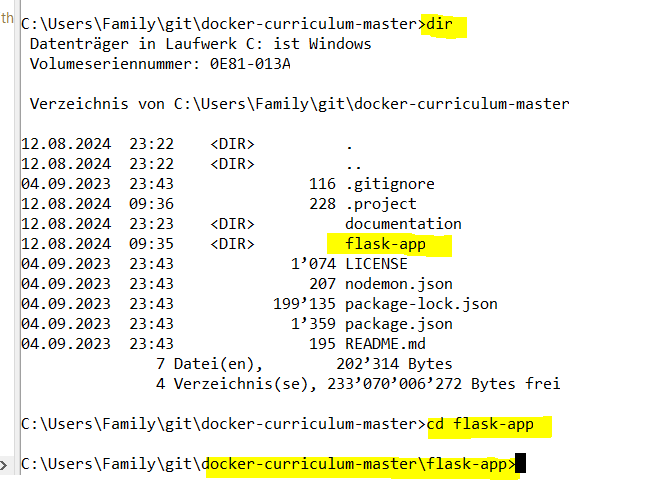
* **Official images** are images that are officially maintained and supported by the folks at Docker. These are typically one word long. In the list of images above, the python, ubuntu, busybox and hello-world images are official images.
* **User images** are images created and shared by users like you and me. They build on base images and add additional functionality. Typically, these are formatted as user/image-name.

Our First Image

Now that we have a better understanding of images, it's time to create our own. Our goal in this section will be to create an image that sandboxes a simple [Flask](http://flask.pocoo.org/) application. For the purposes of this workshop, I've already created a fun little [Flask app](https://github.com/prakhar1989/docker-curriculum/tree/master/flask-app) that displays a random cat .gif every time it is loaded - because you know, who doesn't like cats? If you haven't already, please go ahead and clone the repository locally like so -

$ git clone <https://github.com/prakhar1989/docker-curriculum.git>

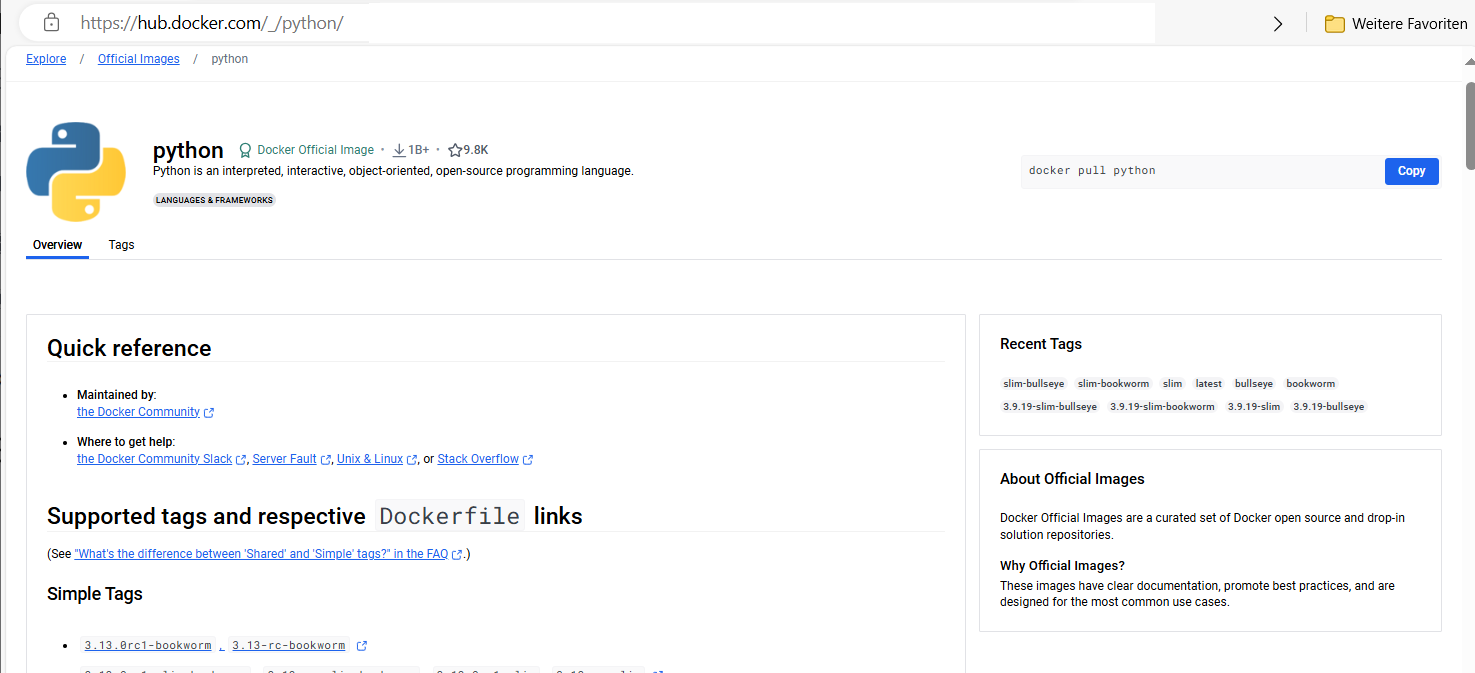
$ cd docker-curriculum/flask-app



*This should be cloned on the machine where you are running the docker commands and not inside a docker container.*

The next step now is to create an image with this web app. As mentioned above, all user images are based on a base image. Since our application is written in Python, the base image we're going to use will

be [Python 3](https://hub.docker.com/_/python/) 🡪 <https://hub.docker.com/_/python/>



Dockerfile

A [Dockerfile](https://docs.docker.com/engine/reference/builder/) is a simple text file that contains a list of commands that the Docker client calls while creating an image. It's a simple way to automate the image creation process. The best part is that the [commands](https://docs.docker.com/engine/reference/builder/#from) 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.

The application directory does contain a Dockerfile but since we're doing this for the first time, we'll create one from scratch. To start, create a new blank file in our favorite text-editor and save it in the **same** folder as the flask app by the name of Dockerfile.

We start with specifying our base image. Use the FROM keyword to do that -

FROM python:3.8

The next step usually is to write the commands of copying the files and installing the dependencies. First, we set a working directory and then copy all the files for our app.

# set a directory for the app

WORKDIR /usr/src/app

# copy all the files to the container

COPY . .

Now, that we have the files, we can install the dependencies.

# install dependencies

RUN pip install --no-cache-dir -r requirements.txt

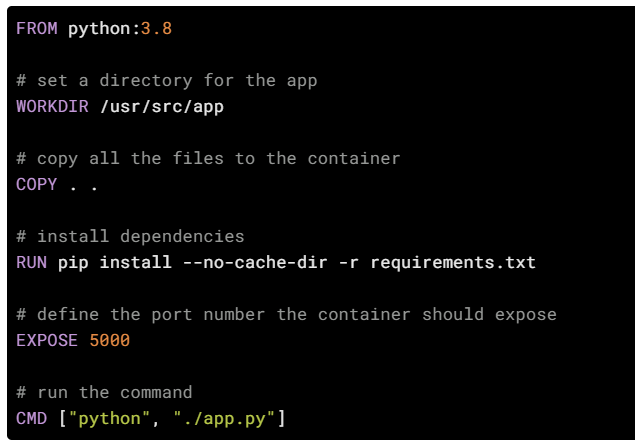
The next thing we need to specify is the port number that needs to be exposed. Since our flask app is running on port 5000, that's what we'll indicate.

EXPOSE 5000

The last step is to write the command for running the application, which is simply - python ./app.py. We use the [CMD](https://docs.docker.com/engine/reference/builder/#cmd) command to do that -

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

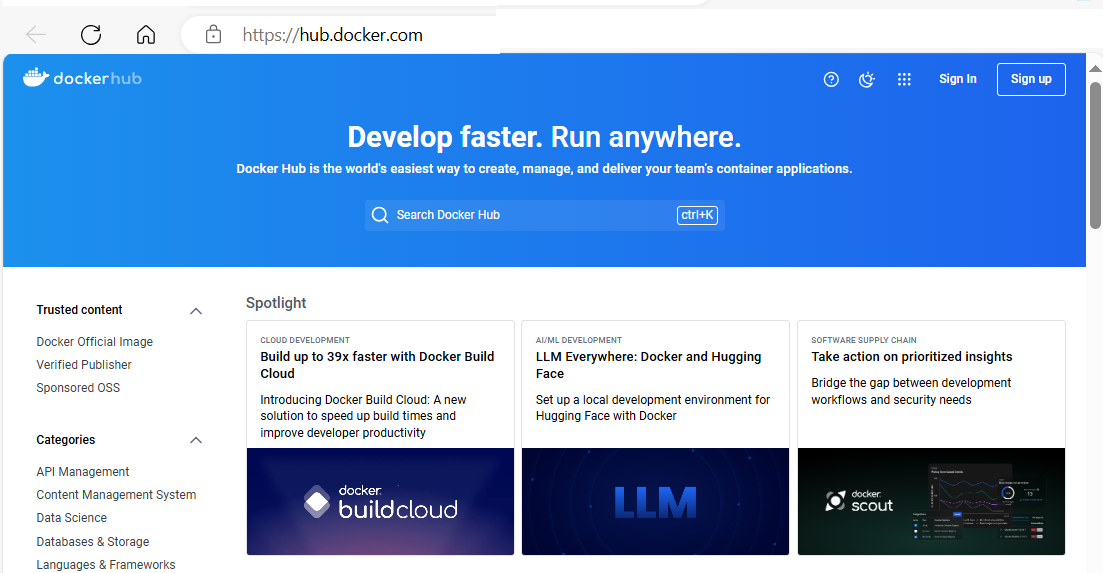
The primary purpose of CMD is to tell the container which command it should run when it is started. With that, our Dockerfile is now ready. This is how it looks –



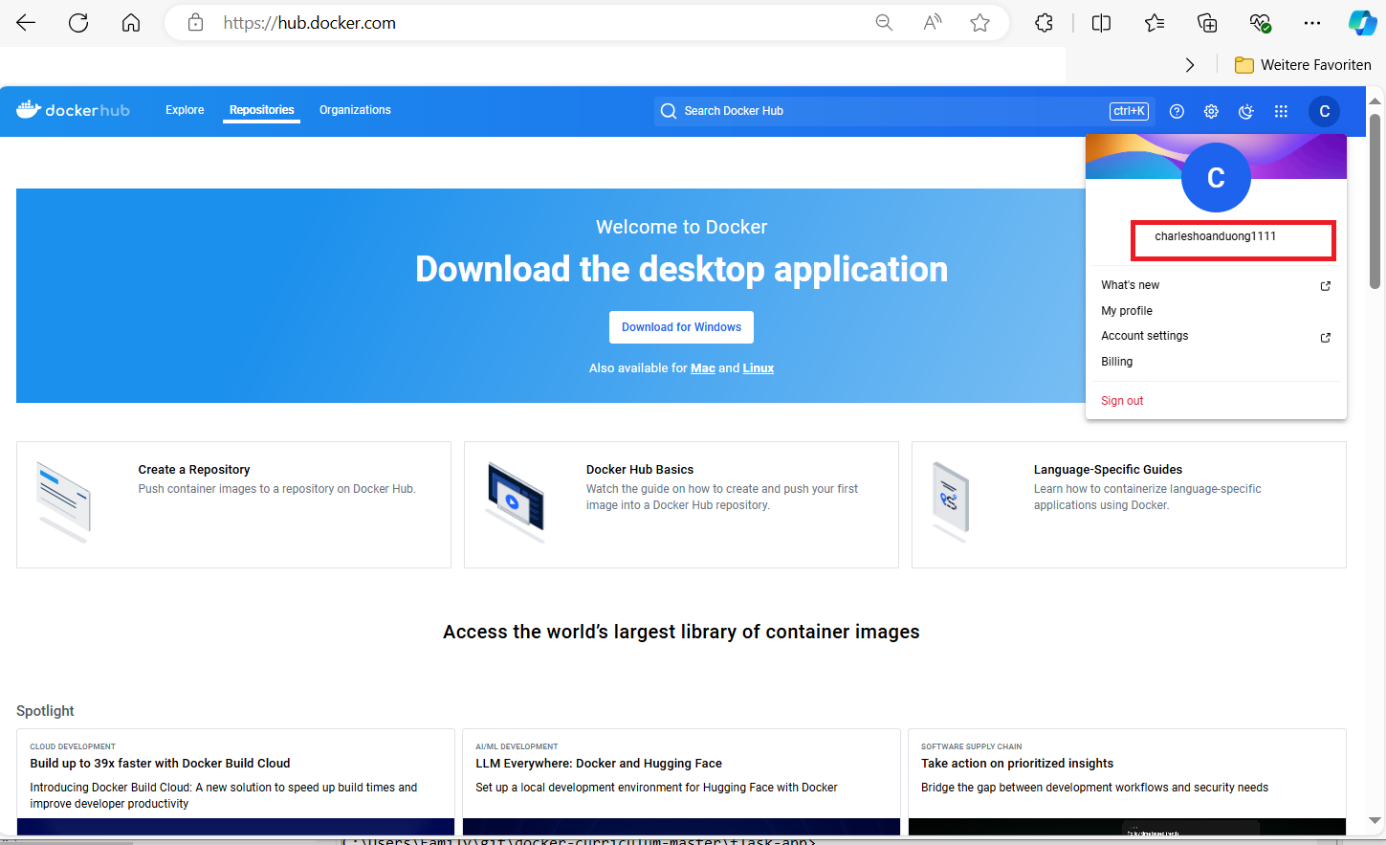
FROM python:3.8  
# set a directory for the app  
WORKDIR /usr/src/app  
# copy all the files to the container  
COPY . .  
# install dependencies  
RUN pip install --no-cache-dir -r requirements.txt  
# define the port number the container should expose  
EXPOSE 5000  
# run the command  
CMD ["python", "./app.py"]

Now that we have our Dockerfile, we can build our image. The docker build command does the heavy-lifting of creating a Docker image from a Dockerfile.

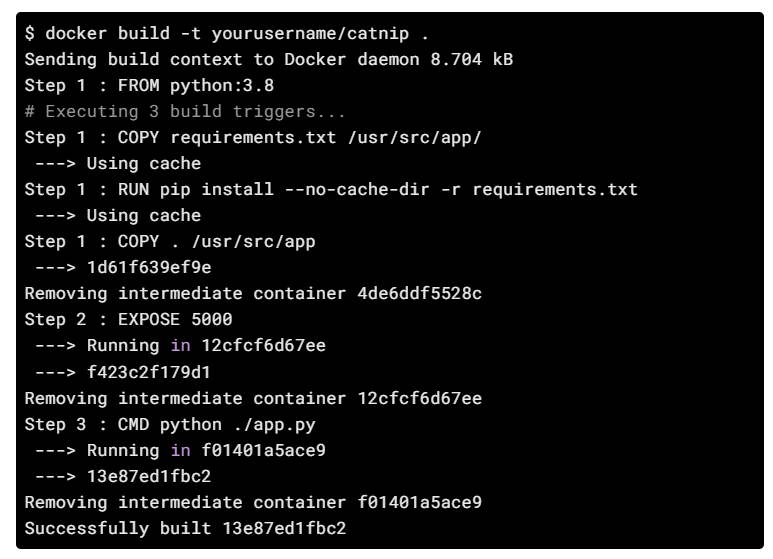
The section below shows you the output of running the same. Before you run the command yourself (don't forget the period), make sure to replace my username with yours. This username should be the same one you created when you registered on [Docker hub](https://hub.docker.com/) 🡪<https://hub.docker.com/>.

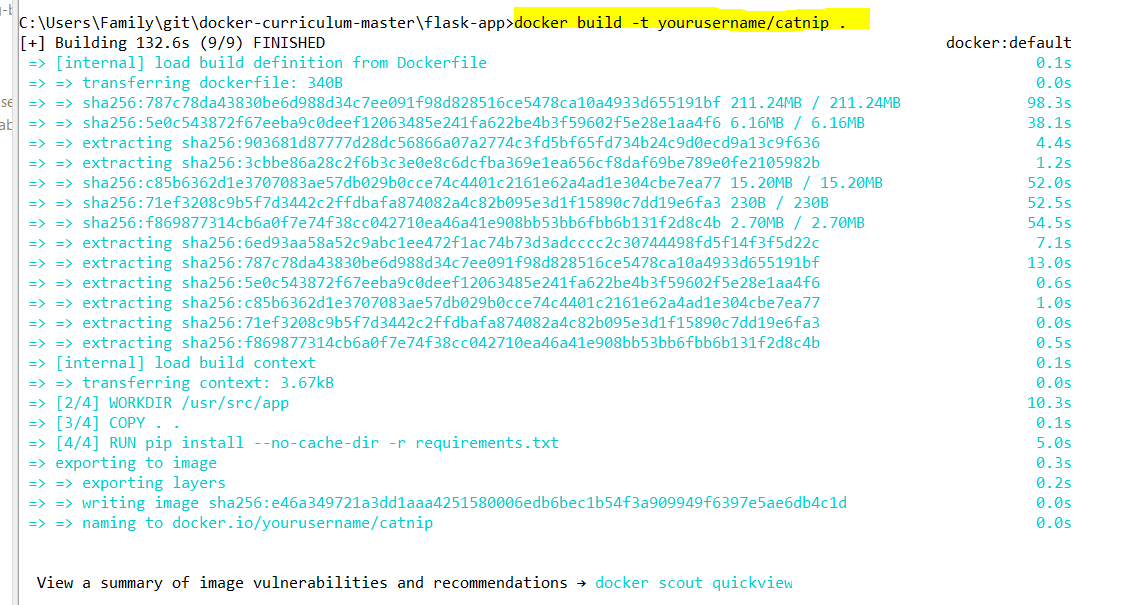


Login Docker Hub with my userid charleshoanduong1111



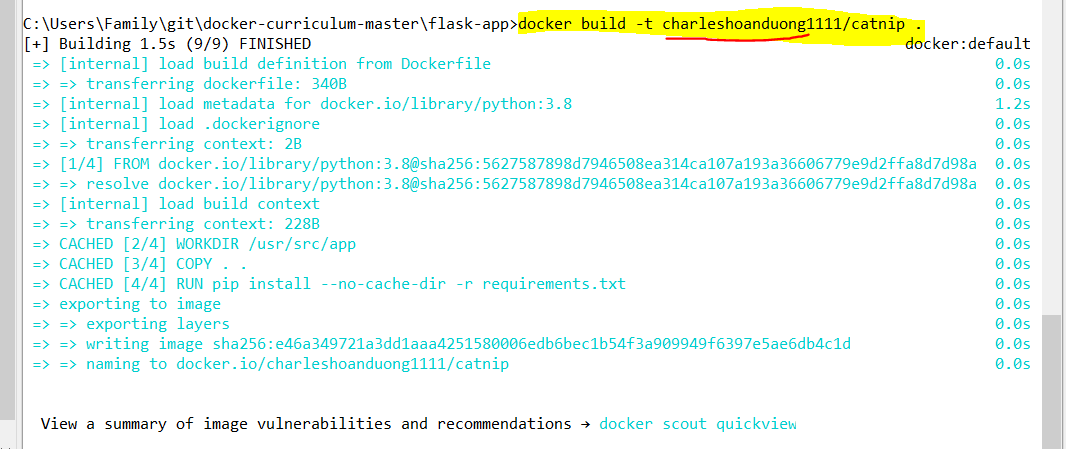
If you haven't done that yet, please go ahead and create an account. The docker build command is quite simple - it takes an optional tag name with -t and a location of the directory containing the Dockerfile.

$ docker build -t yourusername/catnip . 🡪 Sample: Should look as below

C:\Users\Family\git\docker-curriculum-master\flask-app>docker build -t yourusername/catnip . 

Use my Docker Hub userid charleshoanduong1111 instead of yourusername

C:\Users\Family\git\docker-curriculum-master\flask-app>docker build -t charleshoanduong1111/catnip .



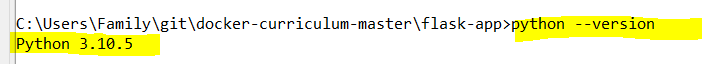
If you don't have the python:3.8 image, the client will first pull the image and then create your image. Hence, your output from running the command will look different from mine. If everything went well, your image should be ready! Run docker images and see if your image shows.

The last step in this section is to run the image and see if it actually works (replacing my username with yours).

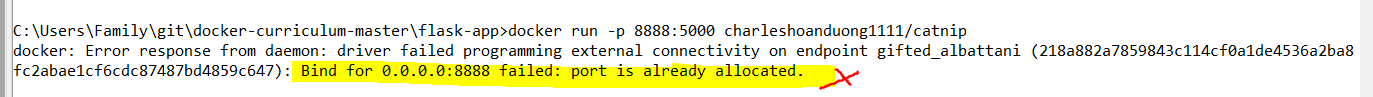
$ docker run -p 8888:5000 yourusername/catnip

\* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

C:\Users\Family\git\docker-curriculum-master\flask-app>python –version



C:\Users\Family\git\docker-curriculum-master\flask-app>docker run -p 8888:5000 charleshoanduong1111/catnip

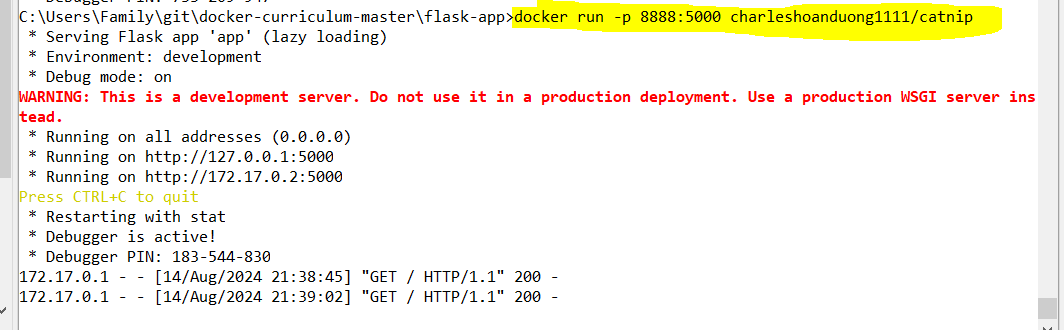


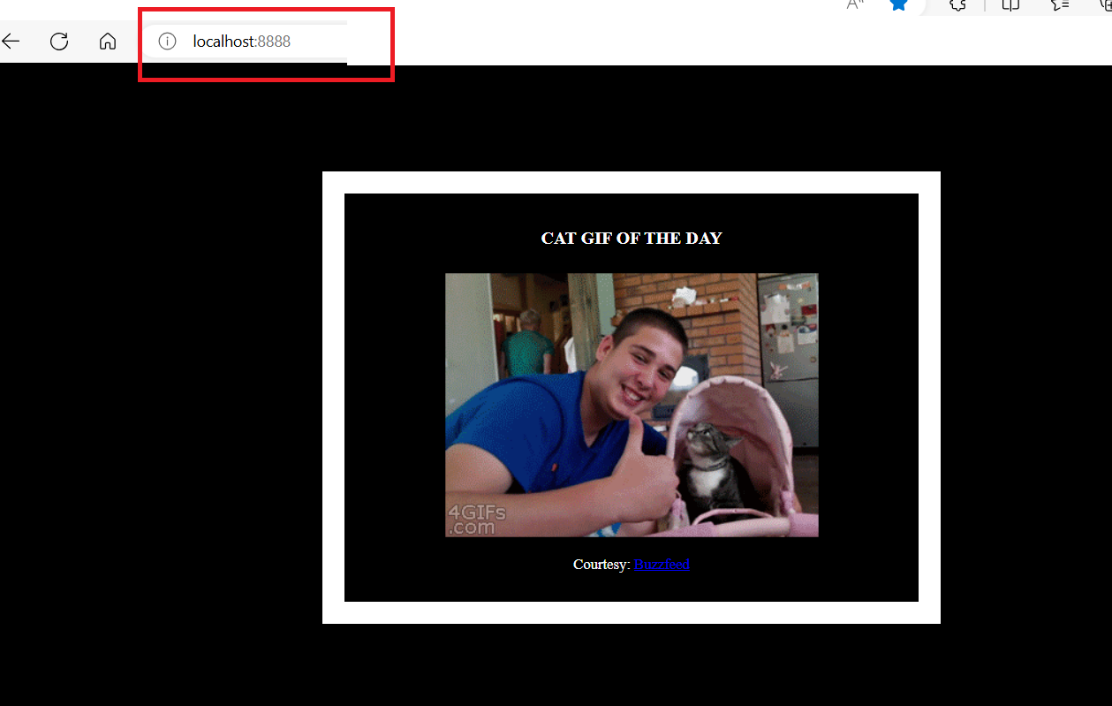
Use Administrator Command to delete the already allocated Port 8888.

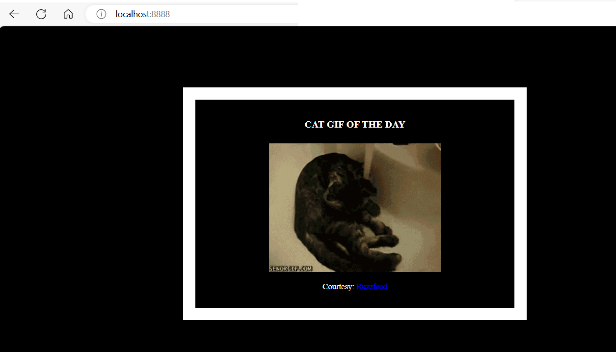
Use 🡪 netstat -ano | findstr :8888 🡪 to find out the PID.

taskkill /PID 29756 /F 🡪 PID=29756 🡪 kill the port 8888





The command we just ran used port 5000 for the server inside the container and exposed this externally on port 8888. Head over to the URL with port 8888, where your app should be live.



Congratulations! You have successfully created your first docker image.