Εργαστήριο 3
Εισαγωγή στο Docker Engine και στα containers

Στο εργαστήριο αυτό θα στήσετε το Docker Engine, θα εγκαταστήσετε και θα ενεργοποιήσετε διάφορα containers. Θα δημιουργήσετε και το δικό σας image.

Step 1 – Setting up your VM

Getting all the tooling setup on your computer can be a daunting task, but thankfully as Docker has become stable, getting Docker up and running on your favorite OS has become very easy. Until a few releases ago, running Docker on OSX and Windows was quite a hassle. Lately however, Docker has invested significantly into improving the on-boarding experience for its users on these OSes, thus running Docker now is a cakewalk. The *getting started* guide on Docker has detailed instructions for setting up Docker on Mac, Linux and Windows.
Please follow the description here in order to install the latest Docker Engine and containerd on VM (Ubuntu).

Once you are done installing Docker, check if the service is running:

```
$ sudo service docker status
```

If it is active (press q) and test your Docker installation by running the following:

```
$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest:
sha256:31b9c7d48790f0d8c50ab433d9c3b7e17666d6993084c002c2ff1ca0
9b96391d
Status: Downloaded newer image for hello-world:latest

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

The Docker daemon binds to a Unix socket instead of a TCP port. By default that Unix socket is owned by the user root and other users can only access it using `sudo`. The Docker daemon always runs as the root user. If you don't want to preface the docker command with `sudo`, create a Unix group called docker and add users to it. When the Docker daemon starts, it creates a Unix socket accessible by members of the docker group.
To create the docker group and add your user:

1. Create the docker group.
   `$ sudo groupadd docker`
2. Add your user to the docker group.
   `$ sudo usermod -aG docker $USER`
3. Log out and log back in so that your group membership is re-evaluated.

If testing on a virtual machine, it may be necessary to restart the virtual machine for changes to take effect. On a desktop Linux environment such as X Windows, log out of your session completely and then log back in. On Linux, you can also run the following command to activate the changes to groups:

$ newgrp docker

4. Verify that you can run docker commands without sudo.

   $ docker run hello-world

Step 2 – Running your first container

2.1 Playing with Busybox

Now that we have everything setup, it's time to get our hands dirty. In this section, we are going to run a Busybox container on our system and get a taste of the docker run command.
To get started, let's run the following in our terminal:

```
$ sudo docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
e5d9363303dd: Pull complete
Digest:
sha256:c5439d7db88ab5423999530349d327b04279ad3161d7596d2126dfb5
b02bfd1f
Status: Downloaded newer image for busybox:latest
docker.io/library/busybox:latest
```

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.

```
$ docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
busybox       latest    b97242f89c8a   3 weeks ago     4.87MB
hello-world   latest    bf756fb1ae65   16 months ago   13.3kB
```

2.2 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.

```
$ 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.

```
$ docker run busybox echo "hello from busybox"
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.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
```

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

```
$ docker ps -a
CONTAINER ID   IMAGE         COMMAND                  CREATED          STATUS                      PORTS     NAMES
ab8fb4eebf3d   busybox       "echo 'hello from bu…"   3 minutes ago    Exited (0) 3 minutes ago              epic_lumiere
273b407b83de   busybox       "sh"                     4 minutes ago    Exited (0) 4 minutes ago              charming_mirzakhani
e40cf1e1d544   hello-world   "/hello"                 28 minutes ago   Exited (0) 28 minutes ago             dreamy_galileo
```

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:

```
$ docker run -it busybox sh
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
/ # uptime
 08:47:42 up 59 min,  0 users,  load average: 0.40, 0.25, 0.21
```

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. 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 (from `sudo docker ps -a`) and paste them alongside the command.

```
$ docker rm ab8fb4eebf3d 273b407b83de
```

```
ab8fb4eebf3d
273b407b83de
```

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.

```
$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Total reclaimed space: 0B
```

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 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.

Step 3 – Running Webapps as containers

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!

2.1 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 you are going to use is a single-page website that was already created for this demo and is available on the Docker Store as `dockersamples/static-site`. You 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.

```
$ docker run --rm dockersamples/static-site
```

Since the image doesn't exist locally, the client will first fetch the image from the registry and then run the image. 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? We need to stop the container. Open another terminal and run the command:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
768a003d90fa	dockersamples/static-site	"/bin/sh -c 'cd /usr…"	20 seconds ago	Up 19 seconds	80/tcp, 443/tcp	strange_elgamal

in order to find the CONTAINER ID or the NAME of the running container. To stop the 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.

```
$ docker stop strange_elgamal
strange_elgamal
```

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.

```
$ docker run -d -P --name static-site dockersamples/static-site
0f55e00304337b3d0b11660ce0b50860dc2c40a8acf27209ee52cfeea0daf393
```

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:

```
$ docker port static-site
443/tcp -> 0.0.0.0:49153
80/tcp -> 0.0.0.0:49154
```

More specifically, what the `run` command does is start a new container based on the `static-site` image. Docker sets up it's own **internal networking** (with its own set of IP addresses) to allow the Docker daemon to communicate with the VM (ubuntu) and to allow containers within Docker to communicate with one another. This internal network is illustrated below.

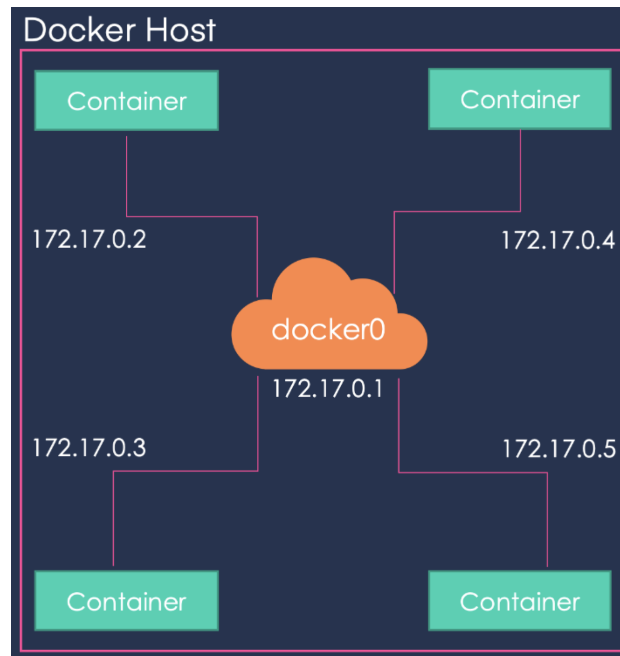
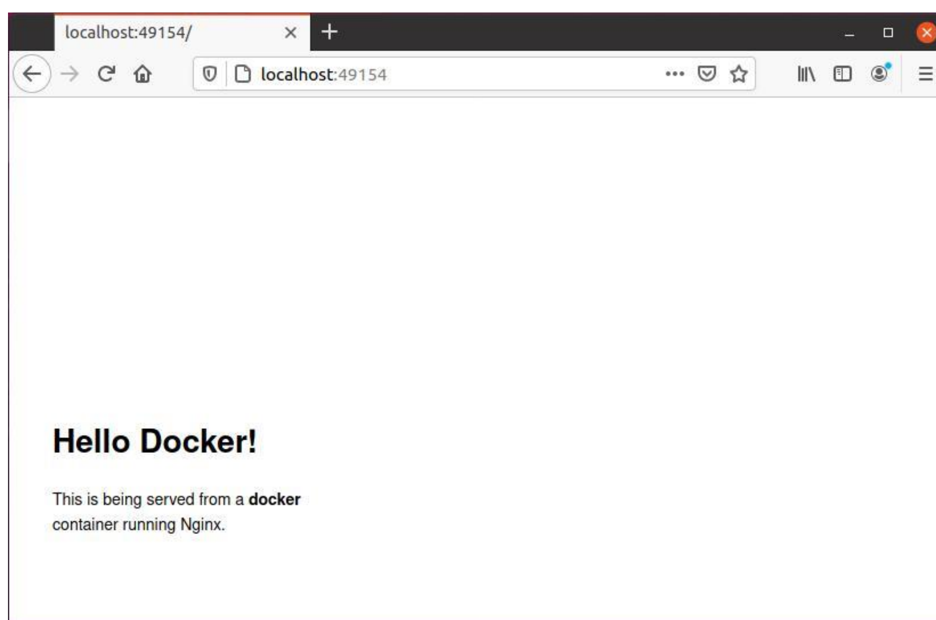


Figure 1: Docker networking. Courtesy of https://towardsdatascience.com/docker-networking-919461b7f498.

As shown above, each container, like our static-site container, has its own IP address. On this container, the Nginx web server is running and serves a static website. The web server on the container opens two ports; port 80 for non-secure (http) connections and port 443 for secure (https) connections. These 2 ports 80 and 443 are not accessible from outside the Docker (i.e. from the VM which is the Docker Host).

So basically, what you're doing with that `-P` is connecting Docker's internal networking with the "external" network - ie. the `-P` option connects random (external) VM ports to internal ports. The `docker port` command reveals that external (VM) port 49153 forwards requests (connects) to internal port 443 and external port 49154 to internal port 80 on the container.

Now, you can open `http://localhost:49154` in your browser.



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

```
$ docker run -d -p 8888:80 -name static-site
dockersamples/static-site
```

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.

```
$ docker stop static-site
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.

Step 4 – 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! 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.

```
$ docker images
REPOSITORY                   TAG       IMAGE ID       CREATED          SIZE
busybox                      latest    b97242f89c8a   3 weeks ago      4.87MB
hello-world                  latest    bf756fb1ae65   16 months ago    13.3kB
dockersamples/static-site    latest    f589ccde7957   6 years ago      191MB
```

The above gives a list of images that we've pulled from the registry. 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 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:

```
$ docker pull ubuntu:20.04
```

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. 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`.

Step 5 – 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 application. For the purposes of this lab, I've already created a fun little 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 virtual 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.

Dockerfile

A Dockerfile 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 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 (e.g. nano) and save it in the **same** folder as the flask app by the name of `Dockerfile`.

```
nano 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.

```
# define the port number the container should expose
EXPOSE 5000
```

The last step is to write the command for running the application, which is simply - python ./app.py. We use the 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"]
```

Use Ctrl-X to save the file and exit the nano editor application.

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 command to create a docker image. Before you run the command yourself (don't forget the period), make sure to replace `yourusername` with yours. This username should be the same one you created when you registered on Docker hub[1]. 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`.

[1] There is no need to create an account to Docker hub. This is required if you want to upload your container to the Docker hub. So you can safely use your ucy username.

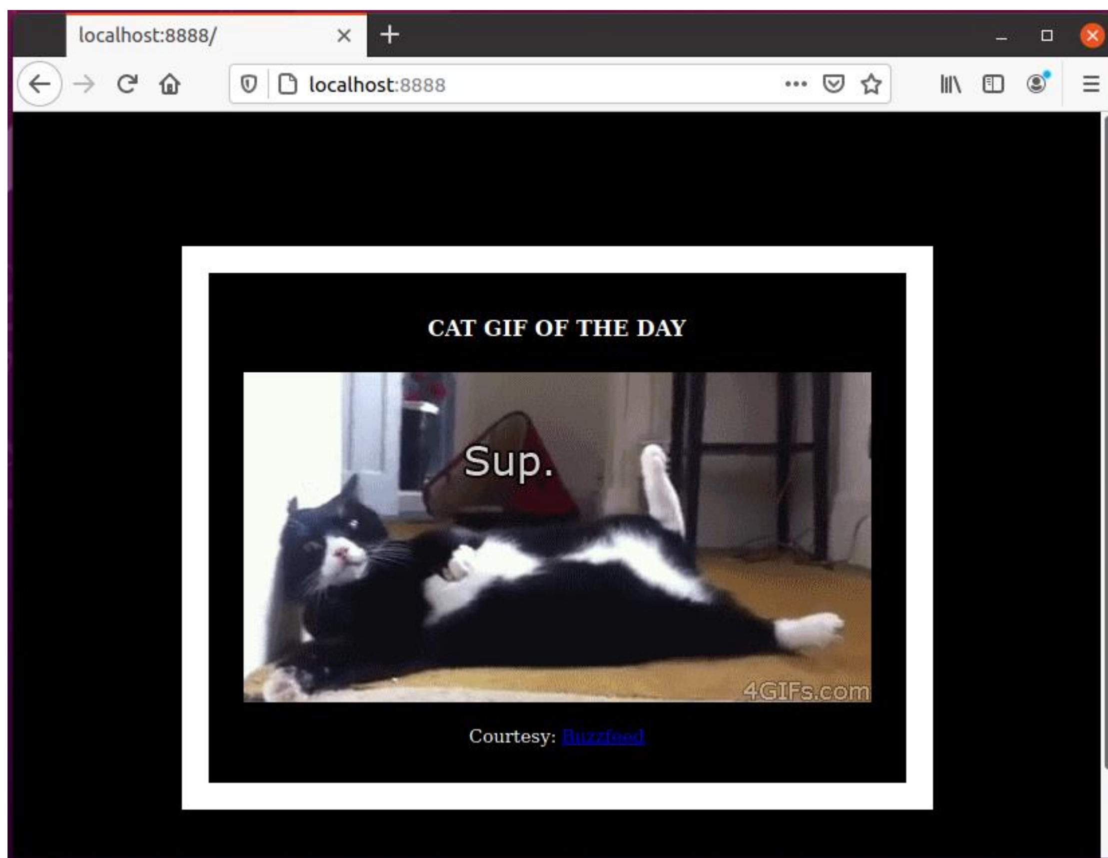
```
$ docker build -t yourusername/catnip .
Sending build context to Docker daemon  8.704kB
Step 1/6 : FROM python:3.8
3.8: Pulling from library/python
bbeef03cda1f: Pull complete
f049f75f014e: Pull complete
56261d0e6b05: Pull complete
9bd150679dbd: Pull complete
5b282ee9da04: Pull complete
03f027d5e312: Pull complete
41b6012c8972: Pull complete
e84941e8ff4e: Pull complete
8848ba81439a: Pull complete
Digest:
sha256:0bdd43369c583eb82a809fbe6908d6ec721b7a29f7d9dce427d70924
baf0ab7b
Status: Downloaded newer image for python:3.8
 ---> e96a212a8ca8
Step 2/6 : WORKDIR /usr/src/app
 ---> Running in 4cf64a344c92
Removing intermediate container 4cf64a344c92
 ---> cb6f9e14a620
Step 3/6 : COPY . .
 ---> 14ccd806395e
Step 4/6 : RUN pip install --no-cache-dir -r requirements.txt
 ---> Running in 77bc9aa9223d
Collecting Flask==2.0.2
  Downloading Flask-2.0.2-py3-none-any.whl (95 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 95.2/95.2 KB 755.6 kB/s eta 0:00:00
Collecting Werkzeug>=2.0
  Downloading Werkzeug-2.2.2-py3-none-any.whl (232 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━ 232.7/232.7 KB 858.6 kB/s eta 0:00:00
Collecting click>=7.1.2
  Downloading click-8.1.3-py3-none-any.whl (96 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 96.6/96.6 KB 2.1 MB/s eta 0:00:00
Collecting Jinja2>=3.0
  Downloading Jinja2-3.1.2-py3-none-any.whl (133 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━ 133.1/133.1 KB 1.9 MB/s eta 0:00:00
Collecting itsdangerous>=2.0
  Downloading itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.1.2-cp38-cp38-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (25 kB)
Installing collected packages: MarkupSafe, itsdangerous, click,
Werkzeug, Jinja2, Flask
Successfully installed Flask-2.0.2 Jinja2-3.1.2 MarkupSafe-
2.1.2 Werkzeug-2.2.2 click-8.1.3 itsdangerous-2.1.2
WARNING: Running pip as the 'root' user can result in broken
permissions and conflicting behaviour with the system package
```

```
manager. It is recommended to use a virtual environment
instead: https://pip.pypa.io/warnings/venv
WARNING: You are using pip version 22.0.4; however, version
23.0 is available.
You should consider upgrading via the '/usr/local/bin/python -m
pip install --upgrade pip' command.
Removing intermediate container 77bc9aa9223d
 ---> 4d7e380926b6
Step 5/6 : EXPOSE 5000
 ---> Running in 3f5f18861cfc
Removing intermediate container 3f5f18861cfc
 ---> 6bbde31c52a2
Step 6/6 : CMD ["python", "./app.py"]
 ---> Running in c9919b118462
Removing intermediate container c9919b118462
 ---> ad13ab028790
Successfully built ad13ab028790
Successfully tagged csp5pa1/catnip:latest
```

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 `yourusername` with yours).

```
$ docker run -p 8888:5000 yourusername/catnip
     * Serving Flask app 'app' (lazy loading)
      * Environment: production
        WARNING: This is a development server. Do not use it in
     a production deployment.
        Use a production WSGI server instead.
      * Debug mode: off
     WARNING: This is a development server. Do not use it in a
     production deployment. Use a production WSGI server
     instead.
      * Running on all addresses (0.0.0.0)
      * Running on http://127.0.0.1:5000
      * Running on http://172.17.0.2:5000
     Press CTRL+C to quit
```

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 (and ran) your first docker image.