Week 6. Session 2.
Cloud Bootcamp practical work.
Based on Docker documentation resources.

Build a Python Docker image

Prerequisites

- You understand basic Docker concepts (from Bootcamp Week 6 Session 1).
- You're familiar with the Dockerfile format (from Bootcamp Week 6 Session 2).

Overview

Now that we have a good overview of containers and the Docker platform, let's take a look at building our first image. An image includes everything needed to run an application - the code or binary, runtime, dependencies, and any other file system objects required.

To complete this tutorial, you need the following:

- Recent stable version of Python. Download Python
- Docker running locally. Download and install Docker
- An IDE or a text editor to edit files. Visual Studio Code

Sample application

Let's create a simple Python application using the Flask framework that we'll use as our example. Create a directory in your local machine named `python-docker` and follow the steps below to create a simple web server.

> WARNING: Windows may not recognise `python3` below. If this happens, try `python` instead.
> TIP: on Windows use the standard command prompt after Docker and Python are fully installed.

```
$ cd /path/to/python-docker
$ python3 -m venv .venv
$ source .venv/bin/activate  # WINDOWS use command ".venv/Scripts/activate"
(.venv) $ python3 -m pip install Flask
(.venv) $ python3 -m pip freeze > requirements.txt
(.venv) $ touch app.py  # WINDOWS use "type nul > app.py"
```

Now, let's add some code to handle simple web requests. Open this working directory in your favorite IDE and enter the following code into the `app.py` file.

```
from flask import Flask
app = Flask(__name__)
```

```
@app.route('/')
def hello_world():
    return 'Hello, Docker!'
```

Test the application

Let's start our application and make sure it's running properly. Open your terminal and navigate to the working directory you created.

```
$ cd /path/to/python-docker
$ source .venv/bin/activate  # WINDOWS use command ".venv/Scripts/activate"
(.venv) $ python3 -m flask run
```

To test that the application is working properly, open a new browser and navigate to `http://localhost:5000`.

Switch back to the terminal where our server is running and you should see the following requests in the server logs. The date and timestamp will be different on your machine.

```
127.0.0.1 - - [20/Feb/2023 11:07:41] "GET / HTTP/1.1" 200 -
```

Create a Dockerfile for Python

Now that our application is running properly, let's take a look at creating a Dockerfile.

> Create an empty file called `Dockerfile` in the `python-docker` directory before proceeding.

Next, we need to add a line in our Dockerfile that tells Docker what base image we would like to use for our application.

```
# syntax=docker/dockerfile:1

FROM python:3.11-slim-buster
```

Docker images can be inherited from other images. Therefore, instead of creating our own base image, we'll use the official Python image that already has all the tools and packages that we need to run a Python application.

To make things easier when running the rest of our commands, let's create a working directory. This instructs Docker to use this path as the default location for all subsequent commands. By doing this, we do not have to type out full file paths but can use relative paths based on the working directory.

```
WORKDIR /app
```

Usually, the very first thing you do once you've downloaded a project written in Python is to install `pip` packages. This ensures that your application has all its dependencies installed.

Before we can run `pip3 install`, we need to get our `requirements.txt` file into our image. We'll use the `COPY` command to do this. The `COPY` command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We'll copy the `requirements.txt` file into our working directory `/app`.

```
COPY requirements.txt requirements.txt
```

Once we have our `requirements.txt` file inside the image, we can use the `RUN` command to execute the command `pip3 install`. This works exactly the same as if we were running `pip3 install` locally on our machine, but this time the modules are installed into the image.

```
RUN pip3 install -r requirements.txt
```

At this point, we have an image that is based on Python version 3.11 and we have installed our dependencies. The next step is to add our source code into the image. We'll use the `COPY` command just like we did with our `requirements.txt` file above.

```
COPY . .
```

This `COPY` command takes all the files located in the current directory and copies them into the image. Now, all we have to do is to tell Docker what command we want to run when our image is executed inside a container. We do this using the `CMD` command. Note that we need to make the application externally visible (i.e. from outside the container) by specifying `--host=0.0.0.0`.

```
CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Here's the complete Dockerfile.

```
# syntax=docker/dockerfile:1

FROM python:3.11-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Directory structure

Just to recap, we created a directory in our local machine called `python-docker` and created a simple Python application using the Flask framework. We also used

the `requirements.txt` file to gather our requirements, and created a Dockerfile containing the commands to build an image. The Python application directory structure would now look like:

```
python-docker
|____ app.py
|____ requirements.txt
|____ Dockerfile
```

Build an image

Now that we've created our Dockerfile, let's build our image. To do this, we use the `docker build` command. The `docker build` command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified PATH or URL. The Docker build process can access any of the files located in this context.

The build command optionally takes a `--tag` flag. The tag is used to set the name of the image and an optional tag in the format `name:tag`. We'll leave off the optional `tag` for now to help simplify things. If you do not pass a tag, Docker uses "latest" as its default tag.

Let's build our first Docker image.

```
$ docker build --tag python-docker .
[+] Building 2.7s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 203B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:3.11-slim-buster
 => [1/6] FROM docker.io/library/python:3.11-slim-buster
 => [internal] load build context
 => => transferring context: 953B
 => CACHED [2/6] WORKDIR /app
 => [3/6] COPY requirements.txt requirements.txt
 => [4/6] RUN pip3 install -r requirements.txt
 => [5/6] COPY . .
 => [6/6] CMD [ "python3", "-m", "flask", "run", "--host=0.0.0.0"]
 => exporting to image
 => => exporting layers
 => => writing image sha256:8cae92a8fbd6d091ce687b71b31252056944b09760438905b726625831564c4c
 => => naming to docker.io/library/python-docker
```

View local images

To see a list of images we have on our local machine, we have two options. One is to use the CLI and the other is to use Docker Desktop. As we are currently

working in the terminal let's take a look at listing images using the CLI.

To list images, simply run the `docker images` command.

```
$ docker images
```

You should see at least one image listed, the image we just built `python-docker:latest`.

Tag images

As mentioned earlier, an image name is made up of slash-separated name components. Name components may contain lowercase letters, digits and separators. A separator is defined as a period, one or two underscores, or one or more dashes. A name component may not start or end with a separator.

> An image is made up of a manifest and a list of layers. Do not worry too much about manifests and layers at this point other than a "tag" points to a combination of these artifacts.

You can have multiple tags for an image. Let's create a second tag for the image we built and take a look at its layers. To create a new tag for the image we've built above, run the following command.

```
$ docker tag python-docker:latest python-docker:v1.0.0
```

The `docker tag` command creates a new tag for an image. It does not create a new image. The tag points to the same image and is just another way to reference the image.

Now, run the `docker images` command to see a list of our local images.

```
$ docker images
```

You can see that we have two images that start with `python-docker`. We know they are the same image because if you take a look at the `IMAGE ID` column, you can see that the values are the same for the two images.

Let's remove the tag that we just created. To do this, we'll use the `rmi` command. The `rmi` command stands for remove image.

```
$ docker rmi python-docker:v1.0.0
```

Note that the response from Docker tells us that the image has not been removed but only "untagged". You can check this by running the `docker images` command.

```
$ docker images
```

Our image that was tagged with `:v1.0.0` has been removed, but we still have the `python-docker:latest` tag available on our machine.

Run your image as a container

> WARNING: the `curl` command used below to test your app will not work on Windows. Use a browser address bar, or the VSCode Thunder Client extension instead.

Above, we created our sample application and then we created a Dockerfile that we used to produce an image. We created our image using the docker command `docker build`. Now that we have an image, we can run that image and see if our application is running correctly.

A container is a normal operating system process except that this process is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.

To run an image inside of a container, we use the `docker run` command. The `docker run` command requires one parameter which is the name of the image. Let's start our image and make sure it is running correctly. Run the following command in your terminal.

```
$ docker run python-docker
```

After running this command, you'll notice that you were not returned to the command prompt. This is because our application is a Flask HTTP server and runs in a loop waiting for incoming requests without returning control back to the OS until we stop the container.

Let's open a new terminal then make a `GET` request to the server using the `curl` command.

> Alternatively, try loading `http://localhost:5000` in your browser's address bar.
> If you are using VSCode, then the Thunder Client extension can also be used.

```
$ curl localhost:5000
curl: (7) Failed to connect to localhost port 5000: Connection refused
```

The `curl` command will fail because the connection to our server was refused. This means, we were not able to connect to the localhost on port 5000. This is expected because our container is running in isolation which includes networking. Let's stop the container and restart with port 5000 published on our local network.

To stop the container, press `ctrl-c`. This will return you to the terminal prompt.

To publish a port for our container, we'll use the `--publish` flag (`-p` for short) on the `docker run` command. The format of the `--publish` command is `[host port]:[container port]`. So, if we wanted to expose port 5000 inside the container to port 3000 outside the container, we would pass `3000:5000` to the `--publish` flag.

We did not specify a port when running the flask application in the container and the default is 5000. If we want our previous request going to port 5000 to work we can map the host's port 8000 to the container's port 5000:

```
$ docker run --publish 8000:5000 python-docker
```

Now, let's rerun the curl command (or browser tab or Thunder Client request) from above. Remember to open a new terminal.

```
$ curl localhost:8000
Hello, Docker!
```

Success! We were able to connect to the application running inside of our container on port 8000. Switch back to the terminal where your container is running and you should see the `GET` request logged to the console.

```
[21/Feb/2023 23:39:31] "GET / HTTP/1.1" 200 -
```

Press `ctrl-c` to stop the container.

Run in detached mode

This is great so far, but our sample application is a web server and we don't have to be connected to the container. Docker can run your container in detached mode or in the background. To do this, we can use the `--detach` or `-d` for short. Docker starts your container the same as before but this time will "detach" from the container and return you to the terminal prompt.

```
$ docker run -d -p 8000:5000 python-docker
ce02b3179f0f10085db9edfccd731101868f58631bdf918ca490ff6fd223a93b
```

Docker started our container in the background and printed the Container ID on the terminal.

Again, let's make sure that our container is running properly. Run the same curl command from above (or use the browser or the Thunder client).

```
$ curl localhost:8000
Hello, Docker!
```

List containers

Since we ran our container in the background, how do we know if our container is running or what other containers are running on our machine? Well, to see a list of containers running on our machine, run `docker ps`. This is similar to how the ps command is used to see a list of processes on a Linux machine.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
ce02b3179f0f        python-docker         "python3 -m flask ru..."   6 minutes ago       Up
```

The `docker ps` command provides a bunch of information about our running containers. We can see the container ID, the image running inside the container, the command that was used to start the container, when it was created, the status, ports that were exposed, and the name of the container.

You are probably wondering where the name of our container is coming from. Since we didn't provide a name for the container when we started it, Docker generated a random name. We'll fix this in a minute, but first we need to stop the container. To stop the container, run the `docker stop` command which does just that, stops the container. You need to pass the name of the container or you can use the container ID.

```
$ docker stop wonderful_kalam
wonderful_kalam
```

Now, rerun the `docker ps` command to see a list of running containers.

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

Stop, start, and name containers

You can start, stop, and restart Docker containers. When we stop a container, it is not removed, but the status is changed to stopped and the process inside the container is stopped. When we ran the `docker ps` command in the previous module, the default output only shows running containers. When we pass the `--all` or `-a` for short, we see all containers on our machine, irrespective of their start or stop status.

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
ce02b3179f0f        python-docker         "python3 -m flask ru..."   16 minutes ago      Exi
ec45285c456d        python-docker         "python3 -m flask ru..."   28 minutes ago      Exi
fb7a41809e5d        python-docker         "python3 -m flask ru..."   37 minutes ago      Exi
```

You should now see several containers listed. These are containers that we started and stopped but have not been removed.

Let's restart the container that we just stopped. Locate the name of the container we just stopped and replace the name of the container below in the restart command.

```
$ docker restart wonderful_kalam
```

Now list all the containers again using the `docker ps` command.

```
$ docker ps --all
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
ce02b3179f0f        python-docker         "python3 -m flask ru..."   19 minutes ago      Up
ec45285c456d        python-docker         "python3 -m flask ru..."   31 minutes ago      Exi
fb7a41809e5d        python-docker         "python3 -m flask ru..."   40 minutes ago      Exi
```

Notice that the container we just restarted has been started in detached mode and has port 8000 exposed. Also, observe the status of the container is "Up X seconds". When you restart a container, it starts with the same flags or commands that it was originally started with.

Now, let's stop and remove all of our containers and take a look at fixing the random naming issue. Stop the container we just started. Find the name of your running container and replace the name in the command below with the name of the container on your system.

```
$ docker stop wonderful_kalam
wonderful_kalam
```

Now that all of our containers are stopped, let's remove them. When you remove a container, it is no longer running, nor it is in the stopped status, but the process inside the container has been stopped and the metadata for the container has been removed.

```
$ docker ps --all
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
ce02b3179f0f        python-docker         "python3 -m flask ru..."   19 minutes ago      Exi
ec45285c456d        python-docker         "python3 -m flask ru..."   31 minutes ago      Exi
fb7a41809e5d        python-docker         "python3 -m flask ru..."   40 minutes ago      Exi
```

To remove a container, run the `docker rm` command with the container name. You can pass multiple container names to the command using a single command. Again, replace the container names in the following command with the container names from your system.

```
$ docker rm wonderful_kalam agitated_moser goofy_khayyam
wonderful_kalam
agitated_moser
goofy_khayyam
```

Run the `docker ps --all` command again to see that all containers are removed.

Now, let's address the random naming issue. Standard practice is to name your containers for the simple reason that it is easier to identify what is running in the container and what application or service it is associated with.

To name a container, we just need to pass the `--name` flag to the `docker run` command.

```
$ docker run -d -p 8000:5000 --name rest-server python-docker
1aa5d46418a68705c81782a58456a4ccdb56a309cb5e6bd399478d01eaa5cdda
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
1aa5d46418a6        python-docker         "python3 -m flask ru..."   3 seconds ago       Up
```

That's better! We can now easily identify our container based on the name.

Bootcamp Tasks

Now that you know how to create a simple Docker image using a Dockerfile, you are ready to extend it with more functionality.

Basic Tasks

- Try running your image in a container that maps port `8080` on the host to port `5000` in the container. Browse to `localhost:8080` to check that it works.
- Change your Flask app code so that it responds with "Hello, Cloud Bootcamp!". Test this works locally on your machine before running a Docker container again. Then, once it works locally, re-run your Flask app Docker container and find out what comes back in the browser. Is it "Hello, Docker!" or "Hello, Cloud Bootcamp!"? Why?
- Rebuild your image and try running a container from it again. What happens now? Why?
- Try `docker run` in two separate terminals with different local ports mapped to the container port `5000`. This way you will have two idential web apps running simultaneously.

Challenge Tasks

- Use the `Dockerfile` to set an environment variable using the `ENV` keyword (look it up in the Dockerfile documentation) so that the variable "SECRET_KEY" has value "12345". Rebuild your image and tag the new version. Use `docker exec` to run an interactive `bash` session *inside the flask application container*. From the container's terminal, check the value of the "SECRET_KEY" environment variable with the command `echo $SECRET_KEY`. Is it what you expect? This approach is the first step towards passing in protected environment information that might change between diffent containers.
- Now consult the `docker run` documentation and work out how to override the `SECRET_KEY` that is in effect inside the container after you execute `docker run` to start the container.
  - TIP: you can do this without rebuilding the image, by using a command line option to `docker run`. Thus the value set in the `Dockerfile` is a default for containers based on the image, but it can be changed by the Docker engine when it starts a new container, if required.
- Finally, extend your existing Flask application image by including the gunicorn package in your `requirements.txt`.
  - TIP 1: get it running locally by running `pip3 install gunicorn` first, then run `pip3 freeze > requirements.txt` in the flask application folder to update the requirements file.
  - TIP 2: you will need to update the `Dockerfile` to use a different `CMD`

command.