1. Docker pull from Docker Hub
2. Docker run
3. Create docke hub account
4. Docker tag…
5. Docker push

\*\* how to pull image from docker hub which have push/created/upload

Docker pull reponame/tabname or image name

We will discussed below topic in today’s session:

1. Create docker file (*Dockerfile*)
2. Install required os/application software/server
3. Create image/build image
4. Run image
5. Manage container
6. Push image to repository (docker hub)
7. Docker image pull into other system

->Create Folder -> cd folder -> create Dockerfile 🡪 write below code -> build docker image -> run docker image

docker build -t friendlyhello . # Create image using this directory's Dockerfile

docker run -p 4000:80 friendlyhello # Run "friendlyname" mapping port 4000 to 80

docker run -d -p 4001:80 friendlyhello # Same thing, but in detached mode

docker container ls # List all running containers

docker container ls -a # List all containers, even those not running

docker container stop <hash> # Gracefully stop the specified container

docker container kill <hash> # Force shutdown of the specified container

docker container rm <hash> # Remove specified container from this machine

docker container rm $(docker container ls -a -q) # Remove all containers

docker image ls -a # List all images on this machine

docker image rm <image id> # Remove specified image from this machine

docker image rm $(docker image ls -a -q) # Remove all images from this machine

docker login # Log in this CLI session using your Docker credentials

docker tag <image> username/repository:tag # Tag <image> for upload to registry

docker push username/repository:tag # Upload tagged image to registry

docker run username/repository:tag # Run image from a registry

Define a container with Dockerfile

Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile behaves exactly the same wherever it runs.

Dockerfile

Create an empty directory. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

* Mkdir myfolder
* Cd myfolder
* Vi/nano/pico/gedit **Dockerfile**

Docker run hello-world

Docker run hello-world

Docker container ps –a

: manage container : stop , start , pause , unpause , kill

-

# Use an official Python runtime as a parent image

FROM Ubuntu

FROM python:2.7-slim

# Set the working directory to /app (folder name)

WORKDIR /app

# Copy the current directory contents into the container at /app

ADD . /app # . current directory

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

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

**Are you behind a proxy server?**

Proxy servers can block connections to your web app once it’s up and running. If you are behind a proxy server, add the following lines to your Dockerfile, using the ENV command to specify the host and port for your proxy servers:

# Set proxy server, replace host:port with values for your servers

ENV http\_proxy host:port

ENV https\_proxy host:port

Add these lines before the call to pip so that the installation succeeds.

This Dockerfile refers to a couple of files we haven’t created yet, namely app.py and requirements.txt. Let’s create those next.

The app itself

Create two more files, requirements.txt and app.py, and put them in the same folder with the Dockerfile. This completes our app, which as you can see is quite simple. When the above Dockerfile is built into an image, app.py andrequirements.txt is present because of that Dockerfile’s ADD command, and the output from app.py is accessible over HTTP thanks to the EXPOSE command.

requirements.txt

Flask

Redis

app.py

from flask import Flask

from redis import Redis, RedisError

import os

import socket

# Connect to Redis

redis = Redis(host="redis", db=0, socket\_connect\_timeout=2, socket\_timeout=2)

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

try:

visits = redis.incr("counter")

except RedisError:

visits = "<i>cannot connect to Redis, counter disabled</i>"

html = "<h3>Hello {name}!</h3>" \

"<b>Hostname:</b> {hostname}<br/>" \

"<b>Visits:</b> {visits}"

return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host='0.0.0.0', port=80)

Now we see that pip install -r requirements.txt installs the Flask and Redis libraries for Python, and the app prints the environment variable NAME, as well as the output of a call to socket.gethostname(). Finally, because Redis isn’t running (as we’ve only installed the Python library, and not Redis itself), we should expect that the attempt to use it here fails and produces the error message.

**Note**: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.

That’s it! You don’t need Python or anything in requirements.txt on your system, nor does building or running this image install them on your system. It doesn’t seem like you’ve really set up an environment with Python and Flask, but you have.

Build the app

We are ready to build the app. Make sure you are still at the top level of your new directory. Here’s what ls should show:

$ ls

Dockerfile app.py requirements.txt

Now run the build command. This creates a Docker image, which we’re going to tag using -t so it has a friendly name.

docker build -t friendlyhello .

Where is your built image? It’s in your machine’s local Docker image registry:

$ docker image ls

REPOSITORY TAG IMAGE ID

friendlyhello latest 326387cea398

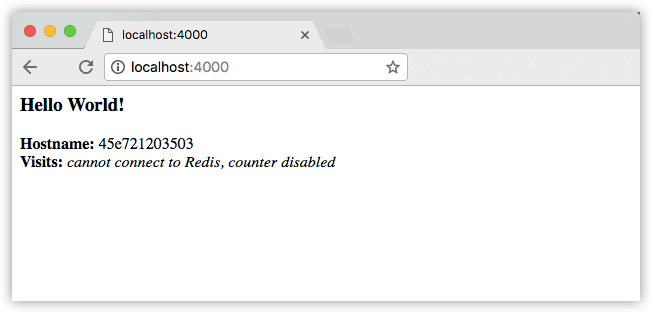
Run the app

Run the app, mapping your machine’s port 4000 to the container’s published port 80 using -p:

docker run -p 4000:80 friendlyhello

You should see a message that Python is serving your app at http://0.0.0.0:80. But that message is coming from inside the container, which doesn’t know you mapped port 80 of that container to 4000, making the correct URL http://localhost:4000.

Go to that URL in a web browser to see the display content served up on a web page.



**Note**: If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of localhost. For example, http://192.168.99.100:4000/. To find the IP address, use the command docker-machine ip.

You can also use the curl command in a shell to view the same content.

$ curl http://localhost:4000

<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b>Visits:</b> <i>cannot connect to Redis, counter disabled</i>

This port remapping of 4000:80 is to demonstrate the difference between what you EXPOSE within the Dockerfile, and what you publish using docker run -p. In later steps, we just map port 80 on the host to port 80 in the container and use http://localhost.

Hit CTRL+C in your terminal to quit.

**On Windows, explicitly stop the container**

On Windows systems, CTRL+C does not stop the container. So, first type CTRL+C to get the prompt back (or open another shell), then type docker container ls to list the running containers, followed bydocker container stop <Container NAME or ID> to stop the container. Otherwise, you get an error response from the daemon when you try to re-run the container in the next step.

Now let’s run the app in the background, in detached mode:

docker run -d -p 4000:80 friendlyhello

You get the long container ID for your app and then are kicked back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker container ls (and both work interchangeably when running commands):

$ docker container ls

CONTAINER ID IMAGE COMMAND CREATED

1fa4ab2cf395 friendlyhello "python app.py" 28 seconds ago

Notice that CONTAINER ID matches what’s on http://localhost:4000.

Now use docker container stop to end the process, using the CONTAINER ID, like so:

docker container stop 1fa4ab2cf395