Your new development environment

In the past, if you were to start writing a Python app, your first order of business was to install a Python runtime onto your machine. But, that creates a situation where the environment on your machine needs to be perfect for your app to run as expected, and also needs to match your production environment.

With Docker, you can just grab a portable Python runtime as an image, no installation necessary. Then, your build can include the base Python image right alongside your app code, ensuring that your app, its dependencies, and the runtime, all travel together.

These portable images are defined by something called a Dockerfile.

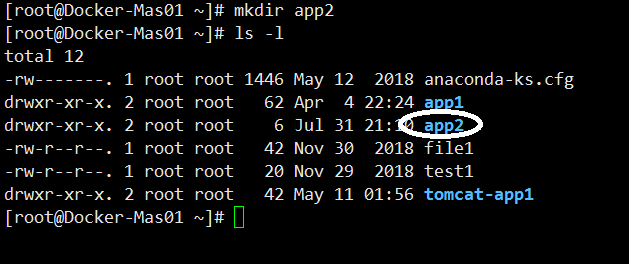
Define a container with Dockerfile

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.

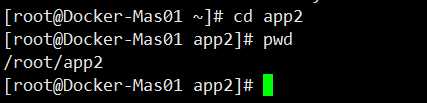
**Create a Folder and create all the below mentioned files inside the folder.**

**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**.

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

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

ADD . /app

# Install any needed packages specified in requirements.txt

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

# Make port 8000 available to the world outside this container

EXPOSE 8000

# Define environment variable

ENV NAME World

# Run app.py when the container launches

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

Press “i” to edit the file

Once copy paste is done

Press “:wq” to save and quit the file.

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 and requirements.txt is present because of that Dockerfile’s ADD command, and the output from app.py is accessible over HTTP thanks to the EXPOSEcommand.

app.py

from flask import Flask

from redis import Redis

import os

import socket

# Connect to Redis

redis = Redis(host="redis", port=6379)

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/>"

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

return 'Hello World! I have been seen {} times.\n'.format(visits)

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

app.run(host='0.0.0.0', port=8000, debug=True)

requirements.txt

flask

redis

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

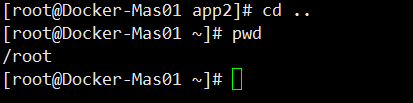
We are ready to build the app. Make sure you are still at the top level of your new directory. Here’s what ls should show:

$ 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.

**Come out of the folder.**



**docker build <foldername> -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

Before proceeding, save daemon.json and restart the docker service.

sudo service docker restart

Once fixed, retry to run the build command.

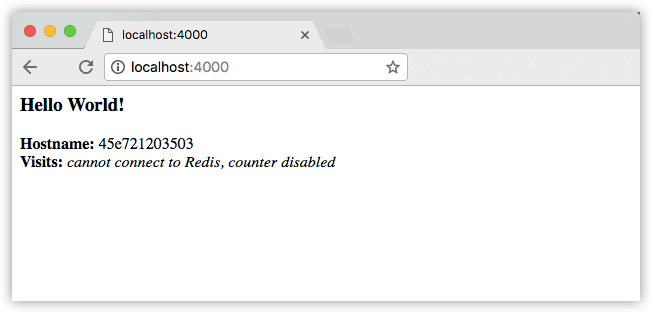
Run the app

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

docker run -p 4000:8000 friendlyhello

You should see a message that Python is serving your app at http://0.0.0.0:8000. But that message is coming from inside the container, which doesn’t know you mapped port 8000 of that container to 4000, making the correct URL http://localhost:8000.

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



**Note: -- “Visits” comes up with an error, that is as expected, because the container is looking for another container called as “Redis” and since its not available the error mssg**

**This is solved with the “DOCKER COMPOSE” feature of Docker**

**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 demonstrates the difference between EXPOSE within the Dockerfile and what the publish value is set to when running docker run -p. In later steps, map port 4000 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 4000:8000 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