Python Docker tutorial

last modified August 25, 2020

Python Docker tutorial shows how to use Docker for Python applications.

Docker

Docker is a platform for developers and sysadmins to build, run, and share applications with containers. Docker facilitates application portability and scalability. Docker provides application isolation and thus eliminates many issues caused by library and environment differences. It helps automate development and deployment. With predefined community images, developers save time and improve their overall experience.

A *Docker image* is a read-only template with instructions for creating a Docker container. A *Docker container* is a runnable instance of an image.

Docker images are stored in repositories. The Docker Hub is the official Docker repository. *Docker Engine* is the underlying client-server technology that builds and runs containers using Docker's components and services.

A *Dockerfile* is a special file which contains instructions needed to build the Docker image.

```
$ sudo docker --version
Docker version 19.03.12, build 48a66213fe
```

This is the Docker version we use.

Python Docker hello example

In the following example, we create and run a very simple Docker image. When we run the image, a simple Python file is executed.

hello.py

```
#!/usr/bin/python

import sys

print("hello there!")
print(sys.version)
```

This is the simple file to be executed inside the container.

Dockerfile

```
FROM python:3.8
COPY hello.py /tmp/
CMD ["python", "/tmp/hello.py"]
```

These are the instructions to build the Docker image.

```
FROM python:3.8
```

We base our image on the community `python:3.8` image.

```
COPY hello.py /tmp/
```

The `COPY` instruction copies the `hello.py` file into the image's `tmp` directory.

```
CMD ["python", "/tmp/hello.py"]
```

The `CMD` instruction lauches the Python program.

```
$ sudo docker build -t hello .
```

We build the image and name it `hello`.

```
$ sudo docker images
REPOSITORY            TAG             IMAGE ID            CREATED             SIZE
hello                 latest          60251c18f538        5 minutes ago       882MB
firstimage            latest          319917db1025        5 hours ago         111MB
python                slim            38cd21c9e1a8        5 days ago          113MB
python                3.8             79cc46abd78d        5 days ago          882MB
python                latest          79cc46abd78d        5 days ago          882MB
```

We list the available images with the `docker images` command.

```
$ sudo docker run hello
hello there!
3.8.5 (default, Aug  5 2020, 08:22:02)
[GCC 8.3.0]
```

We run the `hello` image.

Python Docker interactive mode

We can run the image in interactive mode by using the `-it` option. The `-it` instructs Docker to allocate a pseudo-TTY connected to the container's stdin; creating an interactive bash shell in the container.

```
$ sudo docker run -it python3.8:slim bash
root@98ece0c01946:/#
```

We run the image and get the bash shell inside the container.

```
root@98ece0c01946:/# python -c "import os; print(os.system('ls -l'))"

total 68
drwxr-xr-x   1 root root 4096 Aug  4 16:25 bin
drwxr-xr-x   2 root root 4096 Jul 10 21:04 boot
drwxr-xr-x   5 root root  360 Aug 19 13:40 dev
drwxr-xr-x   1 root root 4096 Aug 19 13:39 etc
drwxr-xr-x   2 root root 4096 Jul 10 21:04 home
drwxr-xr-x   1 root root 4096 Aug  4 16:25 lib
drwxr-xr-x   2 root root 4096 Aug  3 07:00 lib64
drwxr-xr-x   2 root root 4096 Aug  3 07:00 media
drwxr-xr-x   2 root root 4096 Aug  3 07:00 mnt
drwxr-xr-x   2 root root 4096 Aug  3 07:00 opt
dr-xr-xr-x 345 root root    0 Aug 19 13:40 proc
drwx------   1 root root 4096 Aug  4 16:18 root
drwxr-xr-x   3 root root 4096 Aug  3 07:00 run
drwxr-xr-x   2 root root 4096 Aug  3 07:00 sbin
drwxr-xr-x   2 root root 4096 Aug  3 07:00 srv
dr-xr-xr-x  13 root root    0 Aug 19 13:40 sys
drwxrwxrwt   1 root root 4096 Aug 11 20:37 tmp
drwxr-xr-x   1 root root 4096 Aug  3 07:00 usr
drwxr-xr-x   1 root root 4096 Aug  3 07:00 var
0
```

We can run Python code.

```
root@98ece0c01946:/# python -c "import sys; print(sys.version)"
3.8.5 (default, Aug  4 2020, 16:24:08)
[GCC 8.3.0]
```

In our case, we have Python 3.8.5 preinstalled.

Python Docker get request

In the next example, we build an image which retrieves an HTML page. We use the `requests` library, which must be installed in the image.

`get_req.py`

```
#!/usr/bin/python

import requests as req

resp = req.get("http://webcode.me")

print(resp.text)
```

The code example retries a simpe web page by issuing a GET request.

`Dockerfile`

```
FROM python:slim
RUN pip install requests
COPY get_req.py /tmp/
CMD ["python", "/tmp/get_req.py"]
```

This is the `Dockerfile`.

```
FROM python:slim
```

In this example, we use the community `python:slim` image, which occupies much less space.

```
RUN pip install requests
```

With the `RUN` instruction, we execute the `pip` manager and install the `requests` module.

```
COPY get_req.py /tmp/
```

The `COPY` instruction copies the `get_req.py` file from our host computer to the container's `/tmp/` directory. The directory is created if it does not exist.

```
CMD ["python", "/tmp/get_req.py"]
```

The `CMD` instruction runs the program upon starting the container.

```
$ sudo docker build -t pygetreq .
```

We build the image and call it `pygetreq`.

```
$ sudo docker images | grep pygetreq
pygetreq           latest          0d3c5953ff60        23 seconds ago      121MB
```

This image requires only 121MB.

```
$ sudo docker run pygetreq
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My html page</title>
</head>
<body>

    <p>
        Today is a beautiful day. We go swimming and fishing.
    </p>

    <p>
            Hello there. How are you?
    </p>

</body>
</html>
```

This is the output.

Python Docker Flask

In the next example, we run a simple Flask application in a Docker container.

app.py

```
#!/usr/bin/python

from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello there!'
```

This is a simple Flask application with one home route.

Dockerfile

```
FROM python:slim

COPY app.py /app/
WORKDIR /app
RUN pip install flask
RUN export FLASK_APP=app.py

EXPOSE 5000
CMD ["/usr/local/bin/flask", "run", "--host", "0.0.0.0"]
```

This is the Dockerfile.

```
FROM python:slim
```

We base our image on the `python-slim` image.

```
COPY app.py /app/
WORKDIR /app
```

We copy a file to a directory on the image and set the working directory. If the directory does not exist yet, it is created.

```
RUN pip install flask
```

With the `RUN` instruction, we install Flask.

```
RUN export FLASK_APP=app.py
```

We set the `FLASK_APP` environment variable to the `app.py` application file.

```
EXPOSE 5000
```

We expose the port.

```
CMD ["/usr/local/bin/flask", "run", "--host", "0.0.0.0"]
```

With the `CMD` instruction, we set the default command that is run when the container starts. We set the host IP to `0.0.0.0` so that the application is accessible outside the container.

```
$ sudo docker build -t flasksimple .
```

We build the image.

```
$ docker run -p 5000:5000 flasksimple
```

We run the image. The container's 5000 port is mapped to our computer's 5000 port.

```
$ curl localhost:5000/
Hello there!
```

With the `curl` tool, we generate a GET request to the application.

Python Docker MariaDB

In the following example, we create a container based on the `mariadb` image.

schema.sql

```
USE testdb;
DROP TABLE IF EXISTS cities;
CREATE TABLE cities(id INT PRIMARY KEY AUTO_INCREMENT, name VARCHAR(255), population INT);
INSERT INTO cities(name, population) VALUES('Bratislava', 432000);
INSERT INTO cities(name, population) VALUES('Budapest', 1759000);
INSERT INTO cities(name, population) VALUES('Prague', 1280000);
INSERT INTO cities(name, population) VALUES('Warsaw', 1748000);
INSERT INTO cities(name, population) VALUES('Los Angeles', 3971000);
INSERT INTO cities(name, population) VALUES('New York', 8550000);
INSERT INTO cities(name, population) VALUES('Edinburgh', 464000);
INSERT INTO cities(name, population) VALUES('Berlin', 3671000);
```

This is the SQL code for the `cities` table.

Dockerfile

```
FROM mariadb

RUN apt-get update && apt-get install -y \
    python3.8 \
    python3-pip

RUN pip3 install pymysql

ADD schema.sql /docker-entrypoint-initdb.d

ENV MYSQL_USER=user7
ENV MYSQL_PASSWORD=7user
ENV MYSQL_DATABASE=testdb
ENV MYSQL_ROOT_PASSWORD=s$cret

EXPOSE 3306
```

We derive our image from the `mariadb` image.

```
RUN apt-get update && apt-get install -y \
    python3.8 \
    python3-pip
```

On the image, we install Python and pip.

```
RUN pip3 install pymysql
```

We install the `pymysql` driver.

```
ADD schema.sql /docker-entrypoint-initdb.d
```

SQL scripts under `/docker-entrypoint-initdb.d` directory are run at initialization of the container. We copy the schema file with the `ADD` instruction.

```
ENV MYSQL_USER=user7
ENV MYSQL_PASSWORD=7user
ENV MYSQL_DATABASE=testdb
ENV MYSQL_ROOT_PASSWORD=s$cret
```


With environment variables, we create a new user, a new database, and set a password for root user.

```
EXPOSE 3306
```

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime.

app.py

```
#!/usr/bin/python

import pymysql

con = pymysql.connect(host='localhost', user='user7',
            password='7user', database='testdb', port=3306)

try:
    with con.cursor() as cur:

        cur.execute('SELECT * FROM cities')

        rows = cur.fetchall()

        for row in rows:
            print(f'{row[0]}, {row[1]}, {row[2]}')

finally:

    con.close()
```

The example connects to the MariaDB `testdb` database, which is running inside the container. It shows all rows from the `cities` table.

```
$ sudo docker build -t pymaria-simple .
Sending build context to Docker daemon  4.608kB
Step 1/9 : FROM mariadb
    ---> b95867b52886
Step 2/9 : RUN apt-get update && apt-get install -y     python3.8     python3-pip
    ---> Using cache
    ---> 9370769248ed
Step 3/9 : RUN pip3 install pymysql
    ---> Using cache
    ---> 108146aaa2d8
...
```

We build the image.

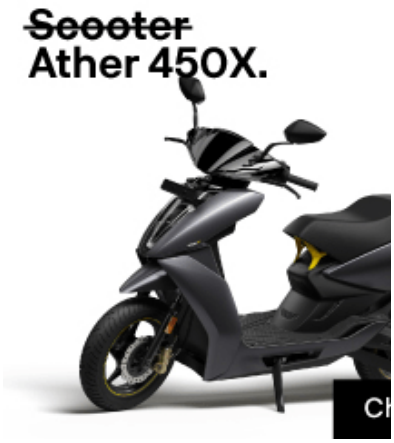
```
$ sudo docker run -p 3306:3306 pymaria-simple
```

We run the container from the custom `pymaria-simple` image. With the `-p` option, we publish a container's 3306 port to the host port 3306.

```
$ ./app.py
1, Bratislava, 432000
2, Budapest, 1759000
3, Prague, 1280000
4, Warsaw, 1748000
5, Los Angeles, 3971000
6, New York, 8550000
7, Edinburgh, 464000
8, Berlin, 3671000
```

After the container is started, we run the `app.py` program, which shows the cities.

In this tutorial, we have worked with Python and Docker.



List all Python tutorials.