

Lab 11 – Containerization



Moving forward with our Docker setup, this week we will take a deeper dive in creating containers by creating an instance of a Docker image. Before moving forward, please give your setup a quick test run to make sure you're all set up by running the following command:

```
`docker run hello-world`
```

When building apps with Docker, the process is typically started at the bottom of the apps architecture which means the process is started with a container.

Portability with Docker

If you've ever had the task of creating a Python app, your first order of business was to install the version runtime onto your local dev environment. Doing this creates a situation where the execution of the script requires the same runtime for it to have the assurance of successfully running if it were ported over to a different environment. Essentially the local developer environment needs to mirror a production environment to assure the application will run as expected.

Docker allows a developer to grab a portable Python runtime as an image with absolutely no installation needed. This creates an environment where the app can build run right alongside the specific version of python chosen and set in the Dockerfile. This ensures that the app has the needed dependencies, and the runtime all travel together when switching to a new environment.

Defining containers with Dockerfile's

A container is defined through a special configuration file unique to Docker. Docker's configuration file is appropriately referred to as a Dockerfile. Dockerfile's define the contents of a container's dependencies – everything from the flavor of Linux that the application is to run, runtime environment specifics, network configurations and so forth. It's with this file that provides the guarantee of identical configurations and runtimes when the Dockerfile is used to produce a container.

Open up a terminal and create a new directory in your home name **lab11_Docker**. Please change to the newly created directory. Using the `touch` command, please create a new file titled '**Dockerfile**'. You can see an image of how I completed this but please take note of the commands used. You'll want to then copy and paste the following:

,

```
# 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
```

```
COPY . /app
```

```
# 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

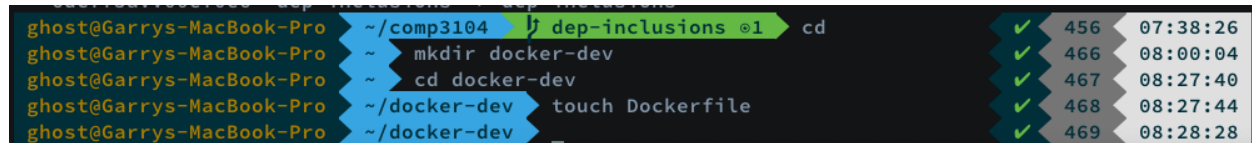
ENV NAME World

Run app.py when the container launches

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

,

As for the commands used to generate the file and what it looks like after pasting the above into my editor, please refer to the below images.



A terminal window screenshot showing the steps to create a Dockerfile. The user is at a Mac with the username 'ghost'. The terminal shows the following commands and their outputs:

Command	Output	Line	Time
<code>cd</code>	<code>~/comp3104</code>	456	07:38:26
<code>mkdir docker-dev</code>	<code>~</code>	466	08:00:04
<code>cd docker-dev</code>	<code>~</code>	467	08:27:40
<code>touch Dockerfile</code>	<code>~/docker-dev</code>	468	08:27:44
	<code>~/docker-dev</code>	469	08:28:28

```
Dockerfile x
docker-dev ▸ Dockerfile ▸ ...
1  # Use an official Python runtime as a parent image
2  FROM python:2.7-slim
3
4  # Set the working directory to /app
5  WORKDIR /app
6
7  # Copy the current directory contents into the container at /app
8  COPY . /app
9
10 # Install any needed packages specified in requirements.txt
11 RUN pip install --trusted-host pypi.python.org -r requirements.txt
12
13 # Make port 80 available to the world outside this container
14 EXPOSE 80
15
16 # Define environment variable
17 ENV NAME World
18
19 # Run app.py when the container launches
20 CMD ["python", "app.py"]
21
```

If you noticed on line 11, we're instructing Docker to run a file that does not exist. On line 20, we are again running Docker syntax to interface with a file that does not exist. Line 11 expects there to be a `requirements.txt` file and line 20 wants there to be an `app.py`. We'll get into the syntax but first, let's create these files and write some code.

Writing our containerized application

We're going to be creating two more files, `requirements.txt` and `app.py`. Let's create these two files in our newly created directory where our `Dockerfile` lives. The application source itself is quite simple coming in at under 25 lines. If you're not familiar with Python, one thing to know is that the language uses whitespace to denote blocks. As JavaScript uses curly braces to denote this, the Python interpreter relies on whitespace to delineate blocks of logic.

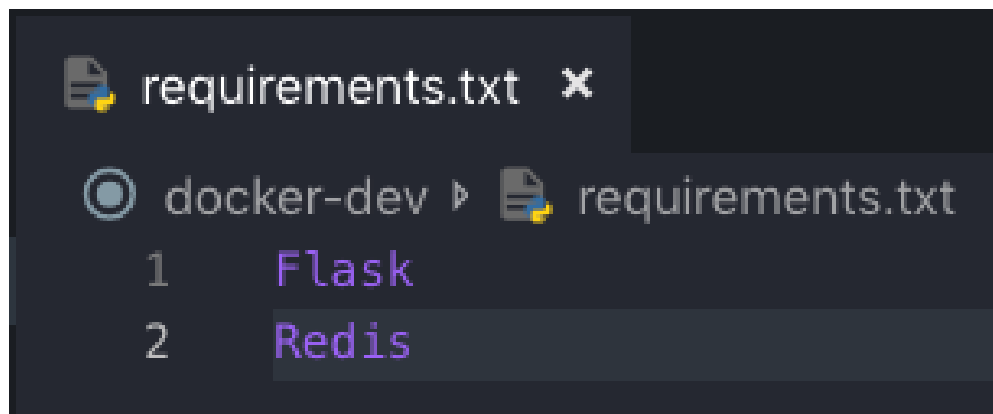
Our requirements file is going to contain our applications requirements, AKA it's dependencies. Which the app only has two, making our requirements file only two lines. Let's start with what we'll be populating our **requirements.txt** file with:

,

Flask

Redis

,

A screenshot of a code editor window. The title bar shows a file icon, the name 'requirements.txt', and a close button. The editor content shows a file explorer on the left with 'docker-dev' selected, and the main area displaying the 'requirements.txt' file with two lines: '1 Flask' and '2 Redis'.

```
requirements.txt x  
docker-dev ▶ requirements.txt  
1 Flask  
2 Redis
```

Please take note of these two lines as you'll see the same keywords found in the source code for our applications dependency import statements. Here is the application source, please not lines 1 and 2:

,

```
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)

```

To clear up any questions of whitespace, here is a screen grab of the source correctly defined in my editor:

```
app.py x
docker-dev ▸ app.py ▸ ...
1  from flask import Flask
2  from redis import Redis, RedisError
3  import os
4  import socket
5
6  # Connect to Redis
7  redis = Redis(host="redis", db=0, socket_connect_timeout=2,
8               socket_timeout=2)
9
10 app = Flask(__name__)
11
12 @app.route("/")
13 def hello():
14     try:
15         visits = redis.incr("counter")
16     except RedisError:
17         visits = "<i>cannot connect to Redis, counter disabled</i>"
18
19     html = "<h3>Hello {name}!</h3>" \
20           "<b>Hostname:</b> {hostname}<br/>" \
21           "<b>Visits:</b> {visits}"
22     return html.format(name=os.getenv("NAME", "world"),
23                       hostname=socket.gethostname(), visits=visits)
24
25 if __name__ == "__main__":
26     app.run(host='0.0.0.0', port=80)
```

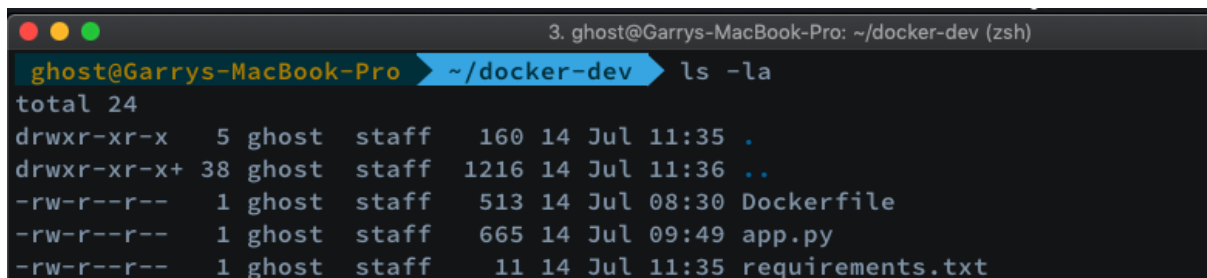
On line 11 of our Dockerfile, we see that ``pip install -r requirements.txt`` command is used to install Python libraries for Flask and Redis. Flask is a lightweight HTTP server used to create web servers and applications with Python. Redis is a caching system commonly used to improve the performance of web applications. Redis works with key/value data structures and stores them in-memory for fast data reads.

On line 21 of our Python script, we are looking for an environment variable called `NAME`, as well as an output of the call made with ``socket.gethostname()`` to retrieve the ID of the container that is running the code. Finally, because Redis isn't running as we haven't installed the service, only a library used to interface with the

service, we should expect the attempt on line 14 to fail and a resulting exception message to be displayed.

Now, we have a Python script and a couple dependencies listed in a requirements file but take notice that we haven't been asked to install Python, Redis or Flask. All we need is a Dockerfile to list our dependencies and on line 2, we can see that Python 2.7 is being used for the image which will be used to create a container instance. The container instance will have Python installed, our host machine doesn't even need to know what Python is. Let's try building the app and see what happens.

Be sure that your terminal is opened in our projects directory and when listed the directory contents, it appears as below with each file containing the contents illustrated above:

A terminal window screenshot showing a directory listing command. The terminal title is '3. ghost@Garrys-MacBook-Pro: ~/docker-dev (zsh)'. The prompt is 'ghost@Garrys-MacBook-Pro' and the command is 'ls -la'. The output shows the contents of the directory: a total size of 24, permissions for '.', '..', 'Dockerfile', 'app.py', and 'requirements.txt', owner 'ghost', group 'staff', sizes, dates, and filenames.

```
3. ghost@Garrys-MacBook-Pro: ~/docker-dev (zsh)
ghost@Garrys-MacBook-Pro ~/docker-dev$ ls -la
total 24
drwxr-xr-x  5 ghost  staff   160 14 Jul 11:35 .
drwxr-xr-x+ 38 ghost  staff  1216 14 Jul 11:36 ..
-rw-r--r--  1 ghost  staff   513 14 Jul 08:30 Dockerfile
-rw-r--r--  1 ghost  staff   665 14 Jul 09:49 app.py
-rw-r--r--  1 ghost  staff    11 14 Jul 11:35 requirements.txt
```

Now that we've confirmed we're in the project directory in our terminal and have our files properly populated, it's time to build our first Docker container, let's run the following command and follow the output:

```
`docker build --tag=hellodocker .`
```

After executing the above command, we should see a success message as shown in the following screengrab which shows that the container was 'Successfully built'.


```
3. ghost@Garrys-MacBook-Pro: ~/docker-dev (zsh)
6847506c074527aa599ec/Click-7.0-py2.py3-none-any.whl (81kB)
Collecting Werkzeug>=0.15 (from Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/9f/57/92a497e38161ce40606c27a86759c6b92dd34fcd
db33f64171ec559257c02/Werkzeug-0.15.4-py2.py3-none-any.whl (327kB)
Collecting itsdangerous>=0.24 (from Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/76/ae/44b03b253d6fade317f32c24d100b3b35c22398
07046a4c953c7b89fa49e/itsdangerous-1.1.0-py2.py3-none-any.whl
Collecting Jinja2>=2.10.1 (from Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/1d/e7/fd8b501e7a6dfe492a433deb7b9d833d39ca749
16fa8bc63dd1a4947a671/Jinja2-2.10.1-py2.py3-none-any.whl (124kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.10.1->Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/fb/40/f3adb7cf24a8012813c5edb20329eb22d5d8e2a
0ecf73d21d6b85865da11/MarkupSafe-1.1.1-cp27-cp27mu-manylinux1_x86_64.whl
Installing collected packages: click, Werkzeug, itsdangerous, MarkupSafe, Jinja2, Flask, Redis
Successfully installed Flask-1.1.1 Jinja2-2.10.1 MarkupSafe-1.1.1 Redis-3.2.1 Werkzeug-0.15.4 click
-7.0 itsdangerous-1.1.0
Removing intermediate container 0ee582cc3610
--> c3bdb18d84bb
Step 5/7 : EXPOSE 80
--> Running in 9906ad55aa6a
Removing intermediate container 9906ad55aa6a
--> d4ae84b2cae9
Step 6/7 : ENV NAME World
--> Running in 885f39aa9a32
Removing intermediate container 885f39aa9a32
--> 85140f27db71
Step 7/7 : CMD ["python", "app.py"]
--> Running in cbd8bd50e679
Removing intermediate container cbd8bd50e679
--> 2decbfc590bc
Successfully built 2decbfc590bc
Successfully tagged hellodocker:latest
ghost@Garrys-MacBook-Pro ~/docker-dev ✓ 474 11:41:27
```

Okay, but where is the instance of our image AKA our container? It's in your machine's local Docker image registry:

```
ghost@Garrys-MacBook-Pro ~/docker-dev docker image ls ✓ 472 11:34:52
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hellodocker	latest	2decbfc590bc	3 minutes ago	148MB

When we executed our build command, we defined a tag name called 'hellodocker'. We can see the tag name in the above image when executing 'docker image ls'.

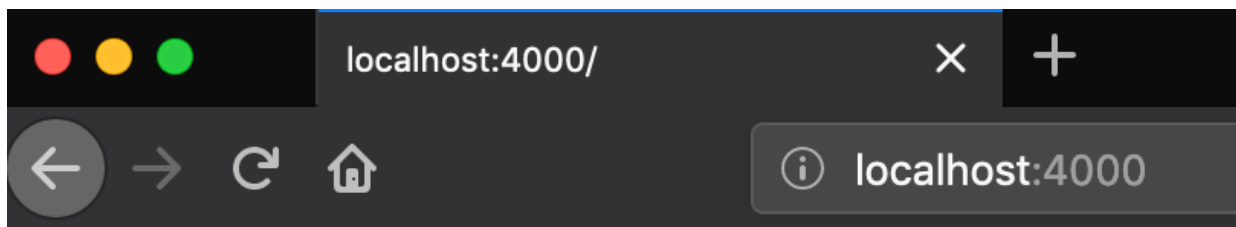
Running our app

The time has come to run our Docker container which contains our Python script, a Python image and our dependencies. The following command is going to interact with our container, using port 80 where the application is hosted and we'll map it to a defined port on our host machine, 4000.

```
`docker run -p 4000:80 hellodocker`
```

```
3. docker run -p 4000:80 hellodocker (docker)
ghost@Garrys-MacBook-Pro ~/docker-dev docker run -p 4000:80 hellodocker ✓ 477 11:50:30
* 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
* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

To break down the above command a bit, ``docker run`` is used to run a container that is listed in our registry. We are then using the ``-p`` switch to map some ports. The ports being mapped is port 80 from our container where the application is hosted. Port 80 is mapped to port 4000 on our host machine. This means that we should be able to request the Flask application defined in our `app.py` script by requesting `localhost:4000` in a browser on our host machine and be presented with the HTML from our `app.py` script.



Hello World!

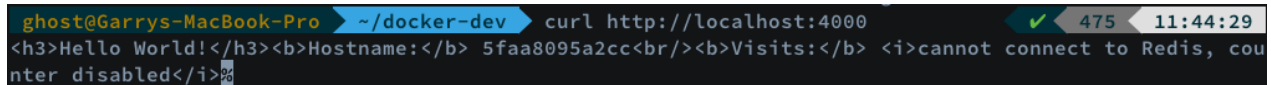
Hostname: 5faa8095a2cc

Visits: *cannot connect to Redis, counter disabled*

We'll also see the Redis counter is not working as we weren't able to increment since we don't have the service installed... yet.

If you're not feeling like opening a browser and wanting to make the request from the command line, don't forget about the power of curl which is very useful when debugging web requests in your terminal.

``curl http://localhost:4000``

A terminal window with a dark background. The prompt is 'ghost@Garrys-MacBook-Pro' followed by a blue arrow pointing to '~/.docker-dev'. The command 'curl http://localhost:4000' is entered. The output is HTML: '<h3>Hello World!</h3>Hostname: 5faa8095a2cc
Visits: <i>cannot connect to Redis, container disabled</i>'. The terminal status bar at the top right shows a green checkmark, '475', and '11:44:29'.

As the above terminal screengrab shows, we weren't able to connect to Redis as the service is not yet set up. In our next lab, we'll see how services can be used in our Docker container.

Before finishing this lab, let's stop our container from running. Go back to your terminal where the `docker run` command was issued on our containers tag name. You should notice some logged web requests if you requested the application hosted on port 80 inside the container.

A terminal window with a dark background. The prompt is 'ghost@Garrys-MacBook-Pro' followed by a blue arrow pointing to '~/.docker-dev'. The command 'docker run -p 4000:80 hellodocker' is entered. The output shows Flask app startup logs: '* 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', and '* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)'. Below this are three log lines for GET requests: '172.17.0.1 - - [14/Jul/2019 15:55:16] "GET / HTTP/1.1" 200 -', '172.17.0.1 - - [14/Jul/2019 15:55:16] "GET /favicon.ico HTTP/1.1" 404 -', and '172.17.0.1 - - [14/Jul/2019 15:57:34] "GET / HTTP/1.1" 200 -'. The terminal status bar at the top right shows a green checkmark, '477', and '11:50:30'.

Let's cancel this by issuing a `ctrl-c` in the terminal running our docker container. If you wanted to ensure the container is no longer running, `docker container ls` can be executed for confirmation. If you see a container ID, you can stop it by running:

``docker container stop CONTAINER_ID``

CONTAINER_ID is to be replaced with the ID outputted when executing ``docker container ls``