Hello Docker World

Hello Flask

In this chapter, we will learn a few things about Docker.

Lets start with a very simple Hello World application in Flask.

For those who do not know, Flask is a minimal web framework for Python. The examples in this book will use Flask. In case you are not familiar with Python or Flask, we will walk you through the code. It should be pretty simple and easy to follow.

As always, its good to get our feet wet with a simple Hello World web application in Flask.

```
# hello.py

# Import the Flask object
from flask import Flask

# Create an object for our web application.
# __name__ in Python is bound to the name of the current module
# which is based on the file name.
# For this case, the name of our file is hello.py so the name of
# our module would be "hello".
app = Flask(__name__)

# This is a decorator. In Python decorators are used to hold
# code that sandwich the code in a function.

# Decorators are used to annotate functions. For instance,
# in the following line, we are saying that the hello function
# should be called whenever the root URL in our web application
# is hit.
@app.route("/")

# This is how you define a function in Python.
def hello():
  # Our function returns the famous "Hello World" string.
  # Like we said the code in the decorator sandwiches the
  # code in our function.
  # We also said that the toast at the top of our sandwich
  # essentially passes control to our function whenever the
  # root of our web application is hit.
  # The toast at the bottom of the sandwich takes whatever
  # the function returns and passes it back to the client
  # that hit the root of our web application.
  return "Hello World!"
```

In order to run this fun little application, we can do the following:

Firstly, we need to install flask. In Python, this is done using pip - which is the Python package manager with a name that Bell Labs would be proud of: PIP Installs Packages.

We can totally install Flask using pip and get on with the rest of it but that would mean Flask would end up in our global Python packages space.

```
$ pip install flask
...

$ which flask
/usr/local/bin/flask

$ 🙄
zsh: command not found: 🤒
```

This is fine. Sometimes. But if you are doing a bunch of work in Python, you would end up with 100s and 1000s of packages on your system. Meh.

Hello Sandbags

The thing that solves this problem is called virtual environments or venv in the Python world.

Virtual environments are cool because they let you install packages on a per-project bases without polluting your global Python installation. Virtual environments are also cool because they come built-in with Python 3.

Did I tell you that we will be using Python 3 in this book? Let me use this opportunity to remind you that Python 2 officially hit its end of life in April 2020 🎳

So firstly, lets create a virtual environment for our little Hello World project. You can do it like so:

```
$ python -m venv ~/<path-to-the-virtual-environment>
```

I usually go with:

```
$ python -m venv ~/.venv/k8s
```

P.S. You can run many modules in Python by using the -m flag. I will come back to this shortly.

A virtual environment is nothing but a copy of all the files that are needed to run Python applications. You can see for yourself:

```
$ ls ~/.venv/k8s
drwxr-xr-x   6 alixedi  staff   192B Jul 31 13:01 .
drwxr-xr-x  36 alixedi  staff   1.1K Jul 31 13:01 ..
drwxr-xr-x  12 alixedi  staff   384B Jul 31 13:01 bin
drwxr-xr-x   2 alixedi  staff    64B Jul 31 13:01 include
drwxr-xr-x   3 alixedi  staff    96B Jul 31 13:01 lib
-rw-r--r--   1 alixedi  staff    75B Jul 31 13:01 pyvenv.cfg
```

Or more interestingly:

```
$ ls ~/.venv/k8s/bin
-rw-r--r--  1 alixedi  staff  2244 Jul 31 13:01 activate
-rw-r--r--  1 alixedi  staff  1300 Jul 31 13:01 activate.csh
-rw-r--r--  1 alixedi  staff  2452 Jul 31 13:01 activate.fish
-rwxr-xr-x  1 alixedi  staff   281 Jul 31 13:01 easy_install
-rwxr-xr-x  1 alixedi  staff   281 Jul 31 13:01 easy_install-3.7
-rwxr-xr-x  1 alixedi  staff   263 Jul 31 13:01 pip
-rwxr-xr-x  1 alixedi  staff   263 Jul 31 13:01 pip3
-rwxr-xr-x  1 alixedi  staff   263 Jul 31 13:01 pip3.7
lrwxr-xr-x  1 alixedi  staff     7 Jul 31 13:01 python -> python3
lrwxr-xr-x  1 alixedi  staff    22 Jul 31 13:01 python3 -> /usr/local/bin/python3
```

We can spot the whole cast in there including the python binary and pip.

Now in order to "activate" this virtual environment, we have - the activate script. Did you spot that in the last code snippet? Well done 👏

Here is how you activate a virtual environment:

```
$ source ~/.venv/k8s/bin/activate
(k8s) $ wow!
zsh: command not found: wow!
```

😬

Right. Now that we have the virtual environment set up, we can install flask in it without polluting our global Python installation.

Here is how you would do it:

```
(k8s) $ pip install flask
...
Installing collected packages: MarkupSafe, Jinja2, Werkzeug, click, itsdangerous, flask
Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1 click-7.1.2 flask-1.1.2 itsdangerous-1.1.0
```

In order to check if our virtual environment is doing what it said on the tin. Or what I said on the tin:

```
(k8s) $ which python
~/.venv/k8s/bin/python
```

To double check:

```
(k8s) $ python
Python 3.7.4 (default, Oct 12 2019, 18:55:28)
[Clang 11.0.0 (clang-1100.0.33.8)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask
>>> awesome!
File "<stdin>", line 1
    awesome!
           ^
SyntaxError: invalid syntax
```

(Sorry again)

One final check. Installing flask in a venv would mean that its is not available globally:

```
(k8s) $ deactivate

$ which python
/usr/local/bin/python3

$ python
..
>>> import flask
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'flask'
```

ModuleNotFoundError. Music to me ears 🕺

Onwards. Flask. Web application.

We wrote ourselves an exciting little application in Flask. We should totally run it. Here is how:

```
(k8s) $ FLASK_APP=hello.py flask run
* Serving Flask app "hello.py"
 * 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://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Looks promising. Lets see if it works. In a different terminal:

```
$ curl localhost:5000
Hello World!
```

wOot! 🙌

I managed to write my excitement in the right window this time 💪

Before I move to containers and specifically Docker containers, I want to press home my advantage and introduce one more little magic trick in pip.

So doing a pip install <awesome-package> is nice and all for development but what if you want to distribute your application?

Wait what? We just cobbled together a Hello World and now we are distributing it? What exactly is distribute anyway? Why is it cool?

Well, I agree that our little Hello World thing is not going to get into ycombinator or anything but its still a fully functional - albeit dumb - web application.

The thing with web applications - like all applications - is that they need to be chucked around fairly often. For instance:

1. You might want to put our Hello World application in a code repository like GitHub. Every time someone in your team adds a little feature to it, you might want to run some tests using e.g. GitHub actions. This is called CI and its all the rave. Now in order to run tests on your application, GitHub actions would need to be able to easily install it - along with all its dependencies no?
2. Once you have them features working, you might want to release the latest version of your application to your users. Generally, for web applications, this would mean putting them on a server. Same story here. On the server you would need to reproduce the environment that your web application needs in order to run. AKA install.
3. You got 2 new junior engineers on your team. They need to quickly and easily be able to run your massively complex Hello World application on their local machine in order to build new features and get you that VC 💰

Luckily, pip has a trick or two up its sleeve that lets us reproduce the environment needed by our web application in order to run.

We can do this in 2 easy steps.

In the first step, we will capture the dependencies required by our web application.

```
(k8s) $ pip freeze
click==7.1.2
Flask==1.1.2
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
Werkzeug==1.0.1
```

This gives us a list of all the dependencies and their correct versions that are installed in our virtual environment right now.

Lets put these in a file like so:

```
(k8s) $ pip freeze > requirements.txt
```

Now that our dependencies are in a file, we can commit this to the repository along with the code and voila! we have taken our first step to enable the distribution of our web application to the CI, the server and other developers.

Anything or anyone who wants to run our web application should be able to do so by following some simple steps:

```
$ # Clone the repo and CD into it
$ git clone k8s
$ cd k8s

$ # Create a new venv and activate it
$ python -m venv ~/.venv/k8s
$ source ~/.venv/k8s/bin activate

$ # Install required packages into the venv
$ pip install -r requirements.txt

$ # Run that thing!
$ FLASK_APP=hello.py flask run
```

You can put these steps in a bash script and it would work beautifully. More or less. Until it doesn't.

Time to talk about containers.

Before I do, a little reminder of what a Python virtual environment is.

```
$ ls ~/.venv/k8s
drwxr-xr-x   6 alixedi  staff   192B Jul 31 13:01 .
drwxr-xr-x  36 alixedi  staff   1.1K Jul 31 13:01 ..
drwxr-xr-x  12 alixedi  staff   384B Jul 31 13:01 bin
drwxr-xr-x   2 alixedi  staff    64B Jul 31 13:01 include
drwxr-xr-x   3 alixedi  staff    96B Jul 31 13:01 lib
-rw-r--r--   1 alixedi  staff    75B Jul 31 13:01 pyvenv.cfg
```

It is a directory containing all the Python dependencies that are needed to run a Python application. This includes the python binary, pip etc.

```
$ ls -l ~/.venv/k8s/bin
total 72
-rw-r--r--  1 alixedi  staff  2244 Jul 31 13:01 activate
..
-rwxr-xr-x  1 alixedi  staff   263 Jul 31 13:01 pip
..
lrwxr-xr-x  1 alixedi  staff     7 Jul 31 13:01 python -> python3
```

It also includes the packages that are needed by our application e.g. Flask:

```
$ ls -a ~/.local/share/virtualenvs/head-1st-k8s/lib/python3.7/site-packages
drwxr-xr-x  21 alixedi  staff   672B Jul 31 13:09 .
drwxr-xr-x   3 alixedi  staff    96B Jul 31 13:01 ..
drwxr-xr-x   9 alixedi  staff   288B Jul 31 13:09 Flask-1.1.2.dist-info
drwxr-xr-x   9 alixedi  staff   288B Jul 31 13:09 Jinja2-2.11.2.dist-info
drwxr-xr-x   8 alixedi  staff   256B Jul 31 13:09 MarkupSafe-1.1.1.dist-info
drwxr-xr-x   8 alixedi  staff   256B Jul 31 13:09 Werkzeug-1.0.1.dist-info
..
```

Its kind of like a sandbag. Sandbags are a cool idea. Like most cool ideas, the sandbag idea increases in awesomeness if we turn it all the way up to 10.

This is what containers do. Well not all the way up to 10. That would be virtual machines. Maybe all the way to 8.

First, lets talk about why.

While putting the packages that our application needs (aka dependencies) to run in a directory so that we can distribute our application easily has been really cool.

But we are still in a half-way house here.

You see the copy of the python binary we spotted in the bin folder in our virtual environment still needs a bunch of operating system libraries to run. If you are running a Linux machine, you can observe this like so:

```
(k8s) $ strace -s 2000 -o strace.log python
Python 3.7.4 (default, Oct 12 2019, 18:55:28)
[Clang 11.0.0 (clang-1100.0.33.8)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

(k8s) $ cat strace.log | head -n 10
execve("/home/alixedi/k8s/bin/python", ["python"], 0x7ffee2a890b0 /* 24 vars */) = 0
brk(NULL)                               = 0xc01000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=17010, ...}) = 0
mmap(NULL, 17010, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8719d60000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

So while the Python virtual environment provides nice isolation within the Python universe, it is still relies on being able to make a bunch of calls to operating system libraries (syscalls) that are not bundled inside the virtual environment.

Not quite a sandbox then is it? What if those junior devs are sporting the trendy surface machines running MS Windows?

Hello Containers

This is where containers come in.

Containers provide a sandbox for your process to run in. This sandbox is to Python virtual environment what Messi is to your Sunday league playmaker or for those of you who aren't into football (why?) what Beyonce is to Britney Spears.

Specifically:

- Containers provides file system isolation. A process running in a container cannot access the file system outside of the container.
- Containers provide memory and CPU usage isolation. A process running in a container can only access the memory and CPU that has been assigned to it.
- Container provide namespace isolation i.e. network, process IDs, hostnames, users etc.
- Containers provide security. A process running in a container can only do what it has been permitted to do.

Finally, Docker is a type of a container. The most popular one no less. It also comes with a bunch of nice-to-haves which makes it easy for beginners.

Enough talk. Lets create our first container. Who is excited? ✋

The first thing we need to do is to write the Docker equivalent of our requirements.txt file i.e. a file that lay out what would go into our container. Standard naming convention dictates that we call this file Dockerfile.

A very basic Dockerfile for our application would look like the following:

```
# Every docker container has a base image (More on images in the following paras).
# It is specified using the FROM keyword on the first line of the Dockerfile.
# Python.org conveniently publishes a bunch of base images that we can use to run Python applications.
# We are using the python base image with the tag `alpine`.
# This would give us a Python environment running on Alpine Linux - a light-weight distribution that is popular in the containe
FROM python:alpine
```

```
# This copies our files to the root of our container.
COPY hello.py requirements.txt /

# This is pretty standard pip and we have already covered it.
# The only interesting thing here is that we are choosing not to use venv.
# We totally can but it will be like running an environment inside a more awesome environment so no need.
RUN pip install -r requirements.txt
```

The attentive reader might ask:

Hang on a minute, we are moving straight to requirements.txt

What happened to being able to create a virtual environment by running:

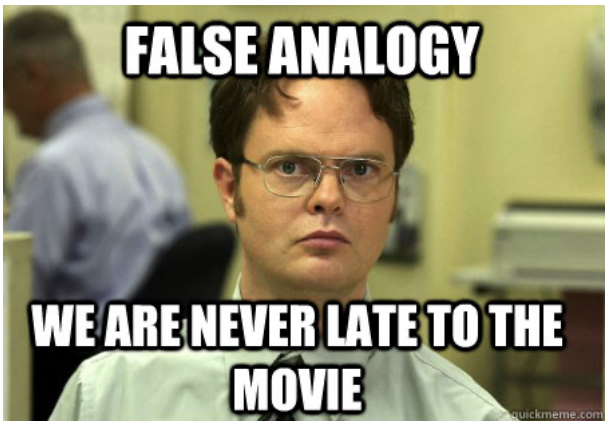
```
$ python -m venv ~/.venvs/k8s
```

Followed by activating the virtual environment and installing a bunch of packages by:

```
$ source ~/.venv/k8s/bin/activate
(k8s) $ pip install flask
```

To which, the clever 🤯 writer would say:

1. Good question.
2. Python venv as the analogy for Docker died as soon as we moved from the What (isolation) to the How (API).



Docker has its own opinions on sandboxes, how they should be built and how processes should be run inside them.

Some of these opinions are informed by the difference in nature of a docker sandbox to a Python venv sandbox.

The clever writer would then move on swiftly to Docker concepts and hope that the analogy with Python venv has caused more good than harm.

Back to Dockerfiles. We have one. What do we do with it?

Without any obvious exceptions, the most awesome thing you could do with a Dockerfile is to use it to produce a Docker image.

What is a Docker image?

It is a tarball that contains all the dependencies needed to run your application.

The act of building a Docker image, among other things, downloads these dependencies from the inter webs onto your machine.

How do you build a docker image?

Simple. Like so:

```
$ docker build .

Sending build context to Docker daemon  5.632kB

Step 1/3 : FROM python:alpine
alpine: Pulling from library/python
df20fa9351a1: Pull complete
36b3adc4ff6f: Pull complete
3e7ef1bb9eba: Pull complete
78538f72d6a9: Pull complete
c9fdb169601a: Pull complete
Digest: sha256:1edaccf11f061da18842e3cb7bb14d9f7336b6b2a24248219ab5c363b8454336
Status: Downloaded newer image for python:alpine
 ---> 872c3118ec53

Step 2/3 : COPY hello.py requirements.txt /
 ---> 70bd00cce758

Step 3/3 : RUN pip install -r requirements.txt
 ---> Running in 9527747772b7
Collecting click==7.1.2
  Downloading click-7.1.2-py2.py3-none-any.whl (82 kB)
Collecting Flask==1.1.2
  Downloading Flask-1.1.2-py2.py3-none-any.whl (94 kB)
Collecting itsdangerous==1.1.0
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting Jinja2==2.11.2
  Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
Collecting MarkupSafe==1.1.1
  Downloading MarkupSafe-1.1.1.tar.gz (19 kB)
Collecting Werkzeug==1.0.1
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Building wheels for collected packages: MarkupSafe
  Building wheel for MarkupSafe (setup.py): started
  Building wheel for MarkupSafe (setup.py): finished with status 'done'
  Created wheel for MarkupSafe: filename=MarkupSafe-1.1.1-py3-none-any.whl size=12629 sha256=7020159b31f20b65caec66087f40ff4a4e
  Stored in directory: /root/.cache/pip/wheels/0c/61/d6/4db4f4c28254856e82305fdb1f752ed7f8482e54c384d8cb0e
Successfully built MarkupSafe
Installing collected packages: click, Werkzeug, MarkupSafe, Jinja2, itsdangerous, Flask
Successfully installed Flask-1.1.2 Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1 click-7.1.2 itsdangerous-1.1.0

Removing intermediate container 9527747772b7
 ---> 1ddb10e93851

Successfully built 1ddb10e93851
```

I have put new lines in the snippet above to distinguish the various steps that were needed for building our docker image.

An attentive reader would notice that the steps correspond to the lines in the Dockerfile. Well done attentive reader. Hold that thought. We will revisit this pretty soon.

Back to the image. We just built one. Where is it though?

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              1ddb10e93851        4 minutes ago       53.4MB
python              alpine              872c3118ec53        2 days ago          42.7MB
```

Hmm. That is not very helpful.

The python image sort of makes sense. I specified it as a base image in my Dockerfile. I presume when I built my image, it got downloaded.

But what about MY image? It doesn't seem to have a REPOSITORY (seems like a name) or a TAG. And what is even a tag anyway?

Lets answer these questions by running the following bit of command-line:

```
$ docker image tag 1ddb10e93851 hello:v1
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello               v1                  1ddb10e93851        11 minutes ago      53.4MB
python              alpine              872c3118ec53        2 days ago          42.7MB
```

Much better.

Except next time, we could bundle up the naming and tagging in the build command itself:

```
$ docker build . --tag hello:v1
```

Now that we know HOW to name and tag our images, its probably useful to understand WHY.

One popular WHY is versions. Learn to take a hint? hello:v1 😉

Imagine a world where every time you release a new version of your app, you build an image and push it to an image-store of sorts.

You would need someway to differentiate hello 1.0 with hello 2.0.

One way to do it would be to name the images hello1 and hello2 respectively.

A better way would be to name the images hello and tag them v1 and v2 respectively.

In docker world, you would refer to these as hello:v1 and hello:v2.

You can also assign multiple tags to an image.

For instance, imagine you test your application in a staging environment before promoting it to production. In this case, it is common to tag the image as staging when it is published. Once you are happy that the thing works, you can assign a production tag to it.

P.S. Assigning staging and production tags wouldn't automatically deploy your application to staging and production environment respectively. There is a missing piece here. We will come back to it when we talk about Kubernetes.

I started on this spiel of names and tags; Conveniently ignoring a small but important detail: There is no name. Its actually repository. Its time we smooth this out.

Now that we know how to build an image, its time to revisit why we started on this path in the first place. We wanted some way to distribute our application in a reproducible way.

Building an images is a strong start but that doesn't quite solve the distribute bit. This is where repositories come in. A repository is a central store for your images. There are several options but for the purpose of learning about repositories, we would run with dockerhub.

Go make an account at Dockerhub. While you are at it, also create a new repository. You can call it hello.

Done? Cool. We can now do the following:

```
$ docker image tag hello:v1 <dockerhub-username>/hello:v1
$ docker push alixedi/hello:v1
The push refers to repository [docker.io/alixedi/hello]
b36cb4db8a04: Pushing [=============>                                     ]   2.84MB/10.76MB
fa9b430b6d13: Pushing [==================================================>]  3.072kB
2e628e2d9dc4: Pushing [==================>                                ]  2.693MB/7.236MB
26e08b3268b4: Pushing [==================================================>]  4.608kB
adf6e7b1c68f: Pushing [======>                                            ]  3.643MB/29.33MB
408e53c5e3b2: Waiting
50644c29ef5a: Waiting
```

This will push the hello:v1 image to dockerhub.

The CI script that deploys your application to production could then do the following:

```
$ docker pull alixedi/hello:v1
```

With a sensible combination of repositories, tags, dockerhub, push and pull, you have just sorted the distribution of your app.

Phew.

But also I realise that we haven't actually run the damned thing yet!

We could run it now but I am inclined to shamelessly double down on the BS and introduce one more concept before that.

This one has been there all along. Looming in the shadows. I have been trying to hold off explaining it. Until now. I am left with no choice. Sorry.

Each instruction in a Dockerfile produces a layer. This layer is a diff of changes in the filesystem from the previous layer. A docker image is a stack of these layers.

One more thing: All the commands produce layers but except for FROM, COPY and RUN, the layers are intermediate. Intermediate layers are squashed in the resulting image.

Finally, how can I inspect these layers? Lets do that now. Break the COPY statement in your Dockerfile:

```
FROM python:alpine
COPY hello.py /
COPY requirements.txt /
RUN pip install -r requirements.txt
```

Build a new version of the image:

```
$ docker build . hello:v2
```

In order to inspect the layers:

```
$ docker history hello:v2
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
5433ece6bdd7        9 minutes ago       /bin/sh -c pip install -r requirements.txt      10.8MB
a320b983dea1        9 minutes ago       /bin/sh -c #(nop) COPY file:11d578baaa9f1fd5…   95B
57d793585c4c        9 minutes ago       /bin/sh -c #(nop) COPY file:35f7847638831fbf…   106B
872c3118ec53        2 days ago          /bin/sh -c #(nop)  CMD ["python3"]              0B
<missing>           2 days ago          /bin/sh -c set -ex;   wget -O get-pip.py "$P…   7.24MB
..
```

And compare the layers with those of hello:v1

```
$ colordiff <(docker history hello:v1) <(docker history hello:v2)
2,3c2,4
< 1ddb10e93851        3 hours ago         /bin/sh -c pip install -r requirements.txt      10.8MB
< 70bd00cce758        3 hours ago         /bin/sh -c #(nop) COPY multi:63205105974020a…   201B
---
> 5433ece6bdd7        11 minutes ago      /bin/sh -c pip install -r requirements.txt      10.8MB
> a320b983dea1        11 minutes ago      /bin/sh -c #(nop) COPY file:11d578baaa9f1fd5…   95B
> 57d793585c4c        11 minutes ago      /bin/sh -c #(nop) COPY file:35f7847638831fbf…   106B
```

Finally, all these layers are not for nothing. Each layer can be re-used by an unlimited number of images. Also, when you are building a new image, Docker is smart enough to only download layers that are not present in the local file system.

Alright, I am now satisfied that we have touched most of the bits in docker that matter.

Time to run that fucking image.

First thing you could do is to run your container with a shell and faff about:

```
$ docker run --interactive --tty hello:v1 sh
```

I would also invite you at this point to summon the help and find out what —interactive and —tty means:

```
$ docker run --help
```

Nice innit? You would be delighted to find out that this help is not limited to docker run.

```
$ docker --help
$ doker build --help
$ docker image --help
$ etc.
zsh: command not found: etc.
```

😐

Back to docker run. Lets try and faff about as promised:

```
$ docker run -it hello:v1 sh
# ls -l -Sr
total 68
dr-xr-xr-x   12 root     root             0 Aug  7 13:16 sys
dr-xr-xr-x  144 root     root             0 Aug  7 13:16 proc
-rw-r--r--    1 root     root            95 Aug  7 09:30 requirements.txt
-rw-r--r--    1 root     root           106 Jul 10 10:14 hello.py
..
```

I see them files and I cannot resist:

```
# FLASK_APP=hello.py flask run &
 * Serving Flask app "hello.py"
 * 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://127.0.0.1:5000/ (Press CTRL+C to quit)

# python
Python 3.8.5 (default, Aug  4 2020, 04:11:56)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import urllib.request
>>> urllib.request.urlopen('http://127.0.0.1:5000').read()
127.0.0.1 - - [07/Aug/2020 13:22:43] "GET / HTTP/1.1" 200 -
b'Hello World!\n'
```

There it is 😅

Now we just want to lift up whatever we did in that container to get our little app running. Here is a decent first go:

```
$ docker run hello:v1 sh -c "FLASK_APP=hello.py flask run"
 * Serving Flask app "hello.py"
 * 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://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Now wouldn't it be fun if we can get that beautiful Hello World back? I am one for trying:

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

There are 2 bits missing here I suspect:

1. The host should not be able to access the localhost for a container. This can be easily fixed by running our app on 0.0.0.0
2. We need some way to forward a port from our host to the container.

Lets see if you can find out how to map ports from the host to the container?

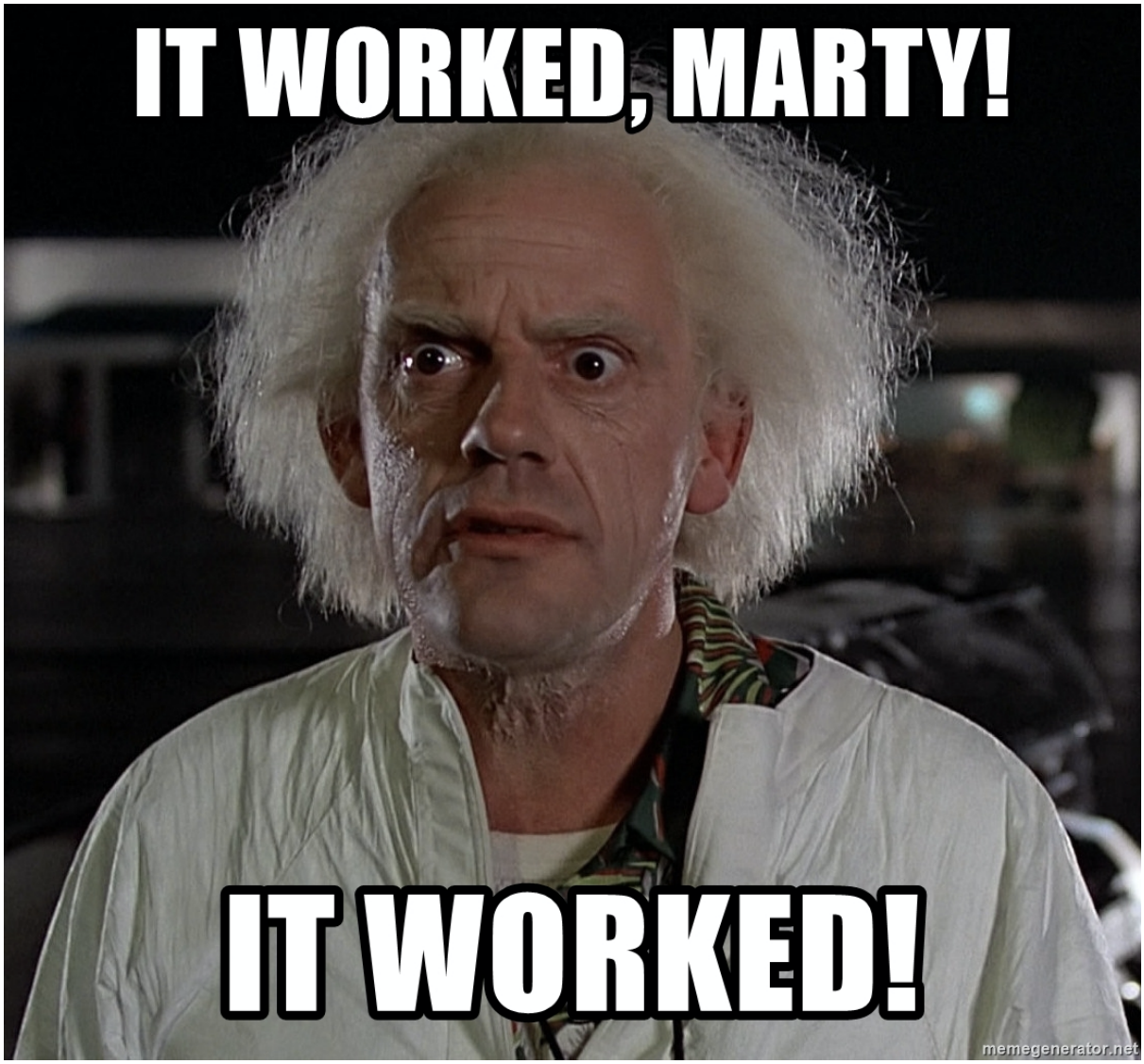
```
$ docker run --help | grep port
      --expose list                    Expose a port or a range of ports
      --health-retries int             Consecutive failures needed to report unhealthy
  -p, --publish list                   Publish a container's port(s) to the host
  -P, --publish-all                    Publish all exposed ports to random ports
```

Looks promising. Lets try the publish bit:

```
$ docker run -p 5000:5000 hello:v1 sh -c "FLASK_APP=hello.py flask run -h 0.0.0.0"
```

Moment of truth ☝️

```
$ curl localhost:5000
Hello World!
```



Now then 🚀

I am taking it as a win but lets be honest. This is a bit of a mouthful:

```
$ docker run -p 5000:5000 hello:v1 sh -c "FLASK_APP=hello.py flask run -h 0.0.0.0"
```

I wonder if we could do something about it?

Turns out we could. Remember that old Dockerfile? Yup. You can totally move a bunch of things in this tedious command line into the Dockerfile.

For starters, you could set environment variables. In addition, you could also specify the command to run when a container is spawned.

Here is how:

```
FROM python:alpine
COPY hello.py requirements.txt /
RUN pip install -r requirements.txt
ENV FLASK_APP=/hello.py
CMD ["flask", "run", "-h", "0.0.0.0"]
```

Lets build and run this baby:

```
$ docker build . hello:v3
$ docker run -p 5000:5000 hello:v3
```

And:

```
$ curl localhost:5000
Hello World!
```

I hope this was fun.

We started with a little Hello World application in Python, introduced Python's virtual environment to make our application **reproducible** and **distributable** (I am actually surprised this is a word).

We then ventured into containers, specifically Docker. We touched upon docker images and how to build them. Finally, we talked about how to run our application inside docker containers.

Using a Hello World application was a bit of a necessity because we wanted complete attention on the container and not what was running inside it. I think we managed to do that but we do not intend to keep at it.