## Overview

Docker is a tool that makes it easy to run our application as a container. In this lab we will:

* Deploy an application in a docker container
* Optimize the container image
* Deploy a second version of the application
* Perform basic troubleshooting

[Overview](#_4l6y9k9n7exo)

[Instructions](#_hxr977vi6sdt)

[Task 0: Explore the docker command](#_hszxfr2ntf23)

[Task 1: Create an application](#_c0i533prk1n7)

[Step 1: Create an App directory](#_y2xjad3y77zo)

[Step 2: Write the application](#_stcldscuqytw)

[Step 3: Run and test the application](#_y8iqsoq879p7)

[Task 2: Containerize the Application](#_gjg0saj5jfxe)

[Step 1: Create Dockerfile](#_qwyzyj22iu7v)

[Step 2: Build image](#_ysv4078yt1m6)

[Step 3: Run the app](#_xni8u8rpim9l)

[Step 4: View the app logs](#_152md5bns5f6)

[Task 3: Optimize the container image](#_lp7wyam4f1)

[Step 1: View image details](#_cv4h48awd8vl)

[Step 2: Change base image](#_9g89q5shkvg2)

[Step 2: Build image](#_ukblld57834y)

[Step 3: View v2 image details](#_6s6eqvl1ba4)

[Task 4: Attempt to deploy 2nd version of app image](#_6yq9bb5j1brn)

[Step 1: Run the app, view port conflict](#_9g418j42ge0)

[Step 2: Cleanup running containers](#_pk2pr9q3g65m)

## 

## Instructions

Read this lab like a book, all text is there for a reason!

"→" denotes an action you must take

Use your favorite editor to edit files within the console. I suggest VI, nano, or emacs.

|  |
| --- |
| White boxes with black text denote commands and file contents |

|  |
| --- |
| Black boxes with green text denote example output |

### Task 0: Explore the docker command

In this lab, we will be using the 'run', 'ps', 'logs', and 'build' commands. There are many other docker commands we will not be making use of, you can read about them with the following command.

→ View the command usage

|  |
| --- |
| docker --help |

For Amazon workspaces, we need to configure docker to properly performed DNS lookups

→ Create file(as sudo) */etc/docker/daemon.json* with the following contents

|  |
| --- |
| {  "dns": ["192.10.0.2", "8.8.8.8"]  } |

→ Restart the docker service

|  |
| --- |
| sudo service docker restart |

### Task 1: Create an application DONE

#### Step 1: Create an App directory

→ Make new directory

|  |
| --- |
| mkdir app  cd app |

#### Step 2: Write the application

→ Open *app.py* in an editor

|  |
| --- |
| vi app.py |

The following app listens on port 8080 and responds to HTTP requests with the value of the RESPONSE environment variable.

→ Type ":set paste", press **enter**. Then press **i**, paste the following:

|  |
| --- |
| import os  from flask import Flask  app = Flask(\_\_name\_\_)  @app.route("/")  def hello():  return os.environ.get('RESPONSE', "No response provided!") + "\n"  if \_\_name\_\_ == "\_\_main\_\_":  app.run(host='0.0.0.0', port=8080) |

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

→ open *requirements.txt* in an editor

|  |
| --- |
| vi requirements.txt |

→ Type ":set paste", press **enter**. Then press **i**, paste the following:

|  |
| --- |
| Flask |

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

#### Step 3: Run and test the application

→ run the app on your workspace

|  |
| --- |
| sudo yum install -y python3 python3-pip  pip3 install --user -r requirements.txt  export RESPONSE="test response"  python3 app.py & |

You should see output similar to:

|  |
| --- |
| \* Serving Flask app "app" (lazy loading)  \* Environment: production  WARNING: Do not use the development server in a production environment.  Use a production WSGI server instead.  \* Debug mode: off  \* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit) |

cURL is a tool we can use the make HTTP requests from the command line, instead of using a web browser.

→ Test the application using curl or open the VM IP in a browser window.

|  |
| --- |
| curl http://localhost:8080 |

→ Terminate the app

|  |
| --- |
| kill %1 |

### Task 1 DONE

### Task 2: Dockerize the Application

#### Step 1: Create Dockerfile

A Dockerfile contains the set of instructions for creating a container image. It tells docker any commands to run in order to install dependencies, how to run the application, and metadata for the image. For more information, see [Dockerfile Reference](https://docs.docker.com/engine/reference/builder/).

**TODO: Makes sure you fill in the dockerfile TODOs below**

→ from within your 'app' directory, open *Dockerfile* in an editor

|  |
| --- |
| vi Dockerfile |

→ Press **i**, fill in your dockerfile:

|  |
| --- |
| FROM ubuntu:20.04  # TODO: Install python3 and python3-pip, hint "apt-get install python3 python3-pip"  # TODO: Copy application files into container  # TODO: Install required python libraries  # TODO: Have the container expose port 8080  CMD python3 /app/app.py |

→ Hit **ESC**, then type ':wq' and press **enter** to save the file.

#### Step 2: Build image

The *docker build* command creates an image using the specified Dockerfile. In this case, it is looking in the current directory, '.', for a file with the name 'Dockerfile'. It will create an image that has the tag "pyapp:v1". Think of a tag as a name for the image, but an image may have multiple tags.

→ Build the image

|  |
| --- |
| docker build -t pyapp:v1 . |

You should see output ending with:

|  |
| --- |
| Step 5/5 : CMD python /app/app.py  ---> Running in 0742747d6ae3  Removing intermediate container 0742747d6ae3  ---> f8ffd102500c  Successfully built f8ffd102500c  Successfully tagged pyapp:v1 |

→ View images that are stored locally:

|  |
| --- |
| docker images |

You should see output similar to the following, show all images that are on this machine.

|  |
| --- |
| REPOSITORY TAG IMAGE ID CREATED SIZE  pyapp v1 2983bcc3c56b 50 seconds ago 510MB  ubuntu 18.04 d131e0fa2585 7 days ago 102MB |

#### Step 3: Run the app

The docker run command will extract the container image and execute the application. Execute the following command to run the image we just created.

→ Run the container

|  |
| --- |
| docker run --name appv1 -e RESPONSE="test response v1" -p 80:8080 -d pyapp:v1 |

|  |
| --- |
| 776e2f1df7047c400ce2694571074c5447863c692561543e0994219609913458 |

In our command, we say that we want port 8080 of the container to be exposed as port 80 on the host vm. We set the RESPONSE environment variable visible to the app.Then finally, we give the running container a name 'appv1' so we can easily refer to it later and add '-d' so it runs detached from our terminal. The output of the command is the unique id of the container.

\*\*NOTE: after I rebooted at the end of the day, I had to run this the next morning with

docker run -e RESPONSE="test response v1" -p 80:8080 -d pyapp:v1

Leaving out the –name portion. The original “docker run” command kept giving me an error about the image name so I sort of accidentally created an appv2, and that’s the one running now.

→ test the app

|  |
| --- |
| curl http://localhost:80 |

|  |
| --- |
| test response v1 |

→ list running containers

|  |
| --- |
| docker ps |

|  |
| --- |
| CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS  NAMES  776e2f1df704 pyapp:v1 "/bin/sh -c 'python3…" 46 seconds ago Up 44 seconds 0.0.0.0:80  ->8080/tcp appv1 |

#### Step 4: View the app logs

Docker makes it easy to access the logs of a container process. Typically, containerized apps log all output to STDOUT and STDERR instead of log files on the filesystem. This is for many reasons, one is that the local filesystem for a container is ephemeral, meaning that if you stop or delete the container you lose those files. Second, if you run an applicatio

n in many places concurrently, it can be difficult to determine which log files to observe when troubleshooting. Many who run containers rely on a log aggregation system (like Splunk or ElasticSearch) to gather this output and make it easily searchable.

→ View the logs of our container

|  |
| --- |
| docker logs appv1 |

|  |
| --- |
| \* Serving Flask app "app" (lazy loading)  \* Environment: production  WARNING: Do not use the development server in a production environment.  Use a production WSGI server instead.  \* Debug mode: off  \* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)  172.17.0.1 - - [04/May/2019 20:06:33] "GET / HTTP/1.1" 200 - |

### Task 2 DONE

### Task 3: Optimize the container image

#### Step 1: View image details

→ View the image

|  |
| --- |
| docker images pyapp:v1 |

As you can see here, our image is a whopping 400+MB. This is because we used a fairly large base image (ubuntu) that contains a lot of packages we do not need. Perhaps instead we can find a smaller base image containing the bare essentials.

#### Step 2: Change base image

Update our Dockerfile to use the Alpine linux distro. Alpine is a lightweight linux distribution, containing many fewer system packages and convenience tools than Ubuntu. We will even take it a step further and use a base image that has python preinstalled.

→ from within your 'app' directory, open *Dockerfile* in an editor

|  |
| --- |
| vi Dockerfile |

→ Type ":set paste", press **enter**. Then press **i**, paste the following:

|  |
| --- |
| FROM python:3.8-alpine  # TODO: Copy application files into container  # TODO: Install required python libraries  # TODO: Have the container expose port 8080  CMD python /app/app.py |

→ Make your changes, hit **ESC**, then type ':wq' and press **enter** to save the file.

We could go even more lightweight than Alpine, even start from ‘scratch’ with no base image. If we did that, however, we would not have any package manager to install python and pip with. This means we would have to do much more work outside of the dockerfile and copy those files into the container, as well as generate a basic linux filesystem structure.

#### Step 2: Build image

→ Build v2 of the image

|  |
| --- |
| docker build -t pyapp:v2 . |

#### Step 3: View v2 image details

→ View the image

|  |
| --- |
| docker images pyapp:v2 |

As you can now see, our new image is only around 80MB! This will be much faster to download to other nodes to run and it does not contain security vulnerabilities that may be introduced by extraneous packages.

**Task 3 DONE**

### Task 4: Attempt to deploy 2nd version of app image

#### Step 1: Run the app, view port conflict

We have decided to test our new image before we kill the container running v1.

→ Run v2

|  |
| --- |
| docker run --name appv2 -e RESPONSE="test response v2" -p 80:8080 -d pyapp:v2 |

|  |
| --- |
| 0af72d9ca4690eb2fc51424d5f9187cbd5d9f550af61f619cfa1f07dc69c3b14  docker: Error response from daemon: driver failed programming external connectivity on endpoint appv2 (4921c74c5ef703af656670858c1d39dd978849b75352e2dd4620239d58047f9b): Bind for 0.0.0.0:80 failed: port is already allocated. |

Uh oh, we can see that we get an error when trying to run v2 of our app alongside v1. “port is already allocated” The first container is using port 8080 of our host already, lets change our v2 to use port 8081 without changing our application from listening on 8080.

→ Run v2

|  |
| --- |
| docker rm appv2  docker run --name appv2 -e RESPONSE="test response v2" -p 8081:8080 -d pyapp:v2 |

|  |
| --- |
| 776e2f1df7047c400ce2694571074c5447863c692561543e0994219609913458 |

→ Test app v2

|  |
| --- |
| curl http://localhost:8081 |

|  |
| --- |
| test response v2 |

Version 1 is also still running in a container

→ Test app v1

|  |
| --- |
| curl http://localhost:80 |

|  |
| --- |
| test response v1 |

#### 

#### Step 2: Cleanup running containers

→ Delete containers

|  |
| --- |
| # Send signal to stop process  docker stop appv1  # Clean up container  docker rm appv1  docker stop appv2  docker rm appv2 |

### Task 4 DONE