Docker and Docker Compose

Software Architecture

March 6, 2023

Brae Webb & Evan Hughes

Figure 1: Docker containers by MidJourney with Prompt ""

1 This Week

This week our goal is to:

- Learn docker basics.
- Dockerise our Todo application from last week.
- Move to a postgresql database from the SQLite database we used last week.
- Testing and Packaging our application on commit.

2 Getting Started with Docker

In the lectures we have delved into the world of containers and how they can be used to create a consistent environment for our applications. Depending on the courses you have taken you may have been exposed to them as either a user or a consumer, for instance if you have submitted assessment to gradescope in CSSE1001 or CSSE2002 the submission system uses docker to run your code.

Notice

This practical will assume you have not used docker before but we will be skipping over some aspects since its not needed for our purposes. Please see the lectures if you would like a better overview to containerisation.

2.1 Docker Images

Images are the building block of the principals on containerisation. They are a self contained environment that can be run that contains the depenedencies and code for an application. A common way to build an image is to use a Dockerfile. Though Docker was not the first to make this concept, it is very popular and relativly easy to use.

2.1.1 Dockerfile

The Dockerfile is a plain text file with its own markup language that is used to describe the image. It is a series of instructions that are run in order to build the image. The instructions are run in a layered fashion, each instruction is run on top of the previous instruction. This means that if you change a line in the Dockerfile you only need to rebuild the layers that have changed. This is a huge time saver when developing your application.

A reference of all possible Dockerfile instructions can be found here: https://docs.docker.com/engine/reference/builder/. For the purposes of the course we are just going to cover some of the basics of [FROM, RUN, COPY, ADD, WORKDIR, CMD, ENTRYPOINT]

FROM The FROM instruction initialises a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a FROM instruction. [1]

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest
```

When building a Dockerfile you need to start with a base image. This is the image that you will build on top of. In this case we are using the latest version of ubuntu. This ubuntu is a very minimal image that is intended to be used as a starting point and is itself an image.

RUN The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile. [1]

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest
```

```
# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip
```

As stated in the definition, the RUN command allows us to perform actions that will happen when the container is built. The example above is installing python dependencies for a fake application. When installing software through a traditional package manager do not forget to update the package list first.

COPY The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>. [1]

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip

# Copying our application into the container
COPY fan-club-runner /app
```

The copy command is one way to pull resources into the container, and is probably the simplest way. This command allows copying individual files. You can also copy entire directories. Be aware that when you COPY files into the container they will exist in the layer forever. The following example will still expose the `secrets.txt` file even though we removed it in another layer.

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip

# Copying our application into the container
COPY fan-club-runner /app

# Removing the secrets file
RUN rm /app/secrets.txt
```

ADD The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>. [1]

ADD can behave like COPY but it can also pull in files and tars from remote locations. This can be handy if your application requires external static files and you do not want to store them in your git repository. Though the ADD command is usually not recommended and is often replaced with curl or wget.

WORKDIR The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile. If the WORKDIR does not exist, it will be created even if it is not used in any subsequent Dockerfile instruction. [1]

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip

# Setting the working directory
WORKDIR /app

# Copying our application into the container
COPY fan-club-runner .
```

The WORKDIR command is a friendly readability feature to help you reduce the amount of shuffling around within the container.

ENTRYPOINT An ENTRYPOINT allows you to configure a container that will run as an executable. [1]

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip

# Setting the working directory
WORKDIR /app

# Copying our application into the container
COPY fan-club-runner .

# Setting the default command
ENTRYPOINT ["python3", "fan-club-runner.py"]
```

The ENTRYPOINT instruction is the executable that will be run when the container starts. The CMD instruction accompanises the ENTRYPOINT and is the default arguments that will be passed to the ENTRYPOINT.

```
docker run test
```

example output:

```
Richard Thomas Fan Club!
```

CMD The main purpose of a CMD is to provide defaults for an executing container. [1]

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip

# Setting the working directory
WORKDIR /app

# Copying our application into the container
COPY fan-club-runner .

# Setting the default command
ENTRYPOINT ["python3", "fan-club-runner.py"]
CMD ["--help"]
```

As stated above the CMD instruction sets default parameters for the ENTRYPOINT but is easily overriden when the container is run. The following example will override the default `--help` argument and run the application with the `--version` argument.

```
docker run test --version
```

example output:

```
Fan Club Runner 1.0.0
```

That concludes the docker instructions we will be looking at so far but there are more that you may need to explore in the future. The full list of instructions can be found in the official documentation [1].

2.1.2 Dockerfile Layers

In the previous sections we used the RUN, ADD and COPY to build up our image to run an example application. When the container is built each of these instructions is run on a layer of the image. This means that if you change a line in the Dockerfile you only need to rebuild the layers that have changed. This is a huge time saver when developing your application.

Let's have a look at an example of the layers that are created when we build our Dockerfile from the previous section.

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip

# Copying our application into the container
COPY . /app
```

```
# Installing our application dependencies
RUN pip3 install -r /app/requirements.txt

# Setting the working directory
WORKDIR /app

# Running our application
CMD ["python3", "src/app.py"]
```

We see that we are copying in the entire application directory before we install our requirements. We can improve this so that we only copy in the requirements file and then install the requirements. This means that if we change our application code we do not need to reinstall the requirements.

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip

# Copying our application into the container
COPY requirements.txt /app/requirements.txt

# Installing our application dependencies
RUN pip3 install -r /app/requirements.txt

# Copying our application into the container
COPY src/ /app

# Setting the working directory
WORKDIR /app

# Running our application
CMD ["python3", "src/app.py"]
```

For the containers we will be building this may not be a large issue, but once your project becomes larger and you have more dependencies it can be a large time saver.

To see the layers that are created when you build your Dockerfile you can use the following command:

```
docker history <image name>

# Example
docker history ubuntu:latest
```

2.1.3 Image Registries

When we introduced the Dockerfile we glossed over the details of from where `ubuntu:latest` is being sourced. This image comes from a registry. There are a few different registries that you can use. The default

registry is one hosted by Docker themselves. This is the registry that is used when you do not specify a registry and is available at https://hub.docker.com/. There are also registries hosted by other companies such as Google and Amazon. These are available at https://cloud.google.com/container-registry and https://aws.amazon.com/ecr/ respectively.

As an example the image that we have been using is available at https://hub.docker.com/_/ubuntu. When browsing this page we can see different tags that are available for this image, such as `latest` and `20.04`. These tags are used to specify different versions of the image so you can pin your image instead of using the `latest` tag, which will always point to the latest version of the image. Note that not all images will have different tags or even a `latest` tag.

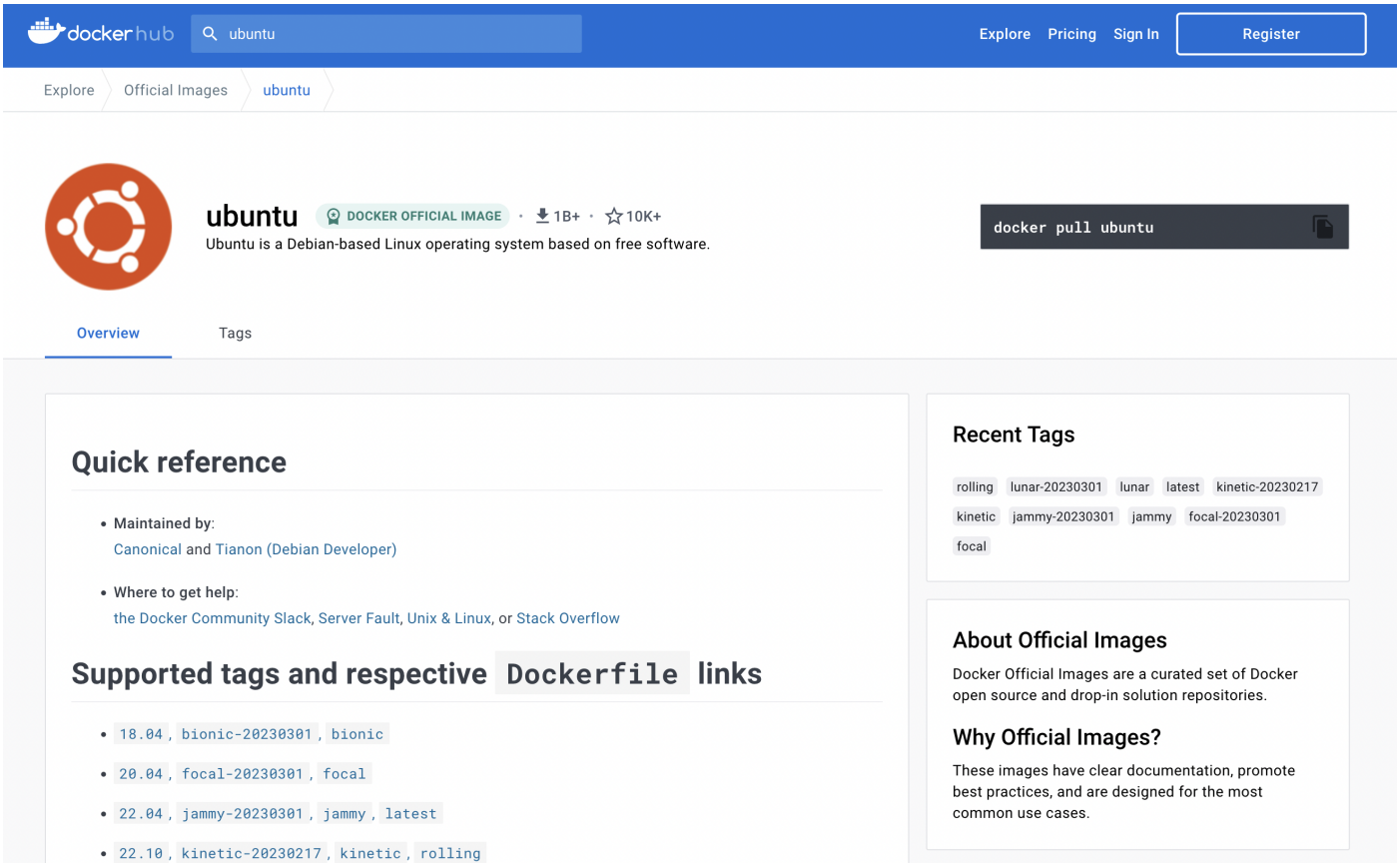

Figure 2: Ubuntu image on Docker Hub

2.2 Docker Containers

TODO: continue fleshing this out

2.2.1 Docker Networking

TODO: continue fleshing this out

2.2.2 Docker Volumes

TODO: continue fleshing this out

2.3 Installing Docker

To install docker on your Mac or Windows machine you can follow the instructions on the docker website: https://docs.docker.com/get-docker/. If you are using a Linux machine you can install docker using your package manager. For example on Ubuntu you can use the following command:

```
sudo apt-get install docker.io
```

> **Info**
>
> If you are using a Linux machine you may need to add your user to the docker group so that you can run docker commands without sudo. While the docker website suggests using docker-desktop for Linux, this course highly recommends using the native docker installation.

2.4 Running a Docker Container

To verify that docker is installed correctly you can run the following command:

```
docker run hello-world

# Example output
...

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

3 Dockerising our Todo Application

3.1 Creating a Practical Repository

Navigate to the GitHub Classroom link for this practical provided by your tutor in Slack. As with last week, this will create a new repository for you in the course organisation. You can now clone this repository to your local machine or work directly in the browser with GitHub codespaces. This repository will be populated with our solution to last week's practical exercise. You may modify this solution or replace it with your own.

3.2 Creating a Dockerfile

Now that we have covered the basics of Docker we can Dockerise our Todo App to prepare it for deployment. Inside our practical folder we will create a new file called `Dockerfile`. To start us off, we will use the following Dockerfile:

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip
```

The above image is what your local environments may have been like when you started the course. To follow the same setup as we have had in the practicals let's install pipenv. We can do this by adding the following line to our Dockerfile:

```
# Install pipenv
RUN pip3 install -y pipenv
```

Let's now change our working directory to `/app` and copy our `Pipfile` and `Pipfile.lock` into the container. We can then install our dependencies using pipenv. We can do this by adding the following lines to our Dockerfile:

```
# Setting the working directory
WORKDIR /app

# Install pipenv dependencies
COPY Pipfile Pipfile.lock /app/
RUN pipenv install --system --deploy
```

Now that we have installed our dependencies we can copy our application into the container. We can do this by adding the following line to our Dockerfile:

```
# Copying our application into the container
COPY todo/ /app
```

Finally we can run our application by adding the following line to our Dockerfile:

```
# Running our application
CMD ["python3", "flask", "--app", "todo", "run", "--host", "0.0.0.0", "--port", "
    6400"]
```

We should now have a complete Dockerfile, as shown below:

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest

# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip

# Install pipenv
RUN pip3 install -y pipenv

# Setting the working directory
WORKDIR /app

# Install pipenv dependencies
```

```
COPY Pipfile Pipfile.lock /app/
RUN pipenv install --system --deploy

# Copying our application into the container
COPY todo/ /app

# Running our application
CMD ["python3", "flask", "--app", "todo", "run", "--host", "0.0.0.0", "--port", "
    6400"]
```

3.3 Building our Docker Image

Now that we have created our Dockerfile we can build our Docker image. To do this we will use the following command:

```
docker build -t todo .

# Example output
...
Successfully tagged todo:latest
```

We can then inspect our layers with the following command:

```
docker image history todo
```

3.4 Running our Docker Container

Now that we have built our Docker image we can run our Docker container. To do this we will use the following command:

```
docker run -p 6400:6400 todo

# Example output
...
* Running on ...
```

We can now navigate to `http://localhost:6400` to see our application running. We can stop our container by pressing `Ctrl+C`. This should show the todo app that we have been working with for the past couple of practicals.

Info

It is important for us to use the host of 0.0.0.0 when running our application. This is because we are running our application inside a container and we need to expose the port to the host machine. If we were to use the default host of 127.0.0.1 we would not be able to access our application from outside the container.

Warning

You do need to make sure that you were not running the webserver from the previous practical on port 6400. If you were you will need to stop that webserver before running the docker container. If you also have other containers running they can conflict with this container.

3.5 Docker Compose

Docker is a useful tool itself but in terms of local development, Docker Compose is a much more useful tool. Docker Compose allows us to run multiple containers together which can be extremely useful if your developing multiple projects or have multiple components that need to be put together. For instance for a story close to home, maybe you are working on INFS2200 with you database schema in one container and you can be running your DECO3801 projects database in another container.

For us, we are going to be using a database closer to our production environment. We will be using a Postgresql database which will eventually be a managed database hosted option from AWS.

3.6 Moving to a Postgresql Database

This week we have actually made a small change to our application so that it can receive the SQLAlchemy database URI as an environment variable. This means that we can now use Docker Compose to run our application and database together. To do this we will create a new file called `docker-compose.yml` and add the following content:

```
version: "3.9"

services:
  database:
    image: postgres:latest
    restart: always
    environment:
      POSTGRES_PASSWORD: verySecretPassword
      POSTGRES_USER: administrator
      POSTGRES_DB: todo
    volumes:
      - ./data:/var/lib/postgresql/data
  app:
    build: .
    restart: always
    environment:
      DATABASE_URI: postgresql://administrator:verySecretPassword@database:5432/
          todo
    ports:
      - "6400:6400"
    depends_on:
      - database
```

We can now run our application and database with the following command:

```
docker-compose up
```

You should observe the output of this command and you can see two processes running. One is the database and the other is our application. We can now navigate to http://localhost:6400 to see our application running. In reality though our database probably did not have enough time to start up before our application tried to connect to it. We can fix this by adding a delay to our application startup. To do this we will add the following line to our Dockerfile:

```
# Adding a delay to our application startup
CMD ["sleep", "10", "&&", "python3", "flask", "--app", "todo", "run", "--host", "
    0.0.0.0", "--port", "6400"]
```

In the long run you would want your application to be able to retry connecting to the database if it fails but for now we will just add a delay. We can now rebuild our Docker image and run our application and database with the following commands:

```
docker-compose up --build
```

You may have noticed that when we defined the todo application service above we did not specify a container. This is because Docker Compose can be instructed to use our Dockerfile to build at time so that we do not have to keep using the build command.

4 Conclusion

Over the past couple of weeks we have built a full todo api that is ready to be deployed to a remote environment. We started with a Flask API returning static responses and now we have a dynamic API that is running in a containerised environment with an external database.

The teaching team do want to emphasise that we have cheated with our docker deployment by using the flask development server. For a production environment you will want to use a production webserver such as gunicorn or uwsgi. These web servers are designed to handle multiple requests at the same time and are much more efficient than the flask development server. We recommend you have a look at how to productionise your flask application as it may help you in your future projects if you choose to use Flask.

References

[1] Docker, "Dockerfile reference." https://docs.docker.com/engine/reference/builder.