

# Docker and Docker Compose

Software Architecture

March 8, 2023

Evan Hughes & Brae Webb



## 1 This Week

This week our goal is to:

- Gain hands on practice with Docker.
- Containerise our Todo application.
- Move to a PostgreSQL database from the SQLite database.

## Notice

## 2 Installing Docker

Before the practical class, please install Docker on your machine. Start the process of installing Docker while your tutor runs through the first part of the practical.

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 recommends using the native docker installation.

### Info

The GitHub Codespaces environment has Docker and Docker-Compose installed.

## 3 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 assignments in CSSE1001 or CSSE2002 the autograding uses Docker to run and test 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 watch the lecture for a foundational understanding of container technology and terminology.

### 3.1 Docker Images

Docker images are the layers of files required to run containers. They are self contained environments that should contain the dependencies and code for any application. A common way to build an image is to use a Dockerfile.

#### 3.1.1 Dockerfile

The Dockerfile is a plain text file with its own syntax that is used to describe the instructions for creating an image. The series of instructions are run in order to create each layer of the image.

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 will cover the core instructions, namely; FROM, RUN, COPY, ADD, WORKDIR, CMD, and ENTRYPOINT.

**FROM** The FROM instruction should be the first instruction in your Dockerfile and specifies the base image for subsequent instructions. If you want to start with an empty image, use the `scratch` image. Otherwise, you likely want to start from an image that has a number of the common utilities pre-packaged.

```
» cat Dockerfile
FROM ubuntu:latest
```

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. Unless you are working in an obscure language, you will often want to start from an image that already has your language or framework installed. For example, you can start from the `nodejs` image, or the `python` image.

**RUN** The RUN instruction will execute commands on top of the current layer and store the changes as a new layer. The layer will be used for the next step in the Dockerfile.

```
» cat Dockerfile
FROM ubuntu:latest

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

The RUN instruction allows us to execute commands when the container is built. The example above is installing python dependencies for a fake application.

#### Info

When using the `apt` package manager on the ubuntu image, remember that the package indexes are removed to reduce the size of the base image. We've made a mistake in the above example resulting in our image size being larger than necessary, can you spot it?

**COPY** The COPY instruction copies files or directories from `<src>` on the host machine where the image is being built to the filesystem of the container at the path `<dest>`.

```
» cat Dockerfile
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 instruction is one of the simpler ways to put resources into the container. Be aware that when you COPY files into the container they will exist in a layer forever. The following example will still expose the `secrets.txt` file even though we removed it in another layer.

```
» cat Dockerfile
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 files, directories, or remote file URLs from `<src>` to the filesystem of the image at the path `<dest>`.

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

```
ADD https://csse6400.uqcloud.net/slides/containers.pdf .
```

```
RUN curl https://csse6400.uqcloud.net/slides/containers.pdf -o containers.pdf
```

**WORKDIR** The WORKDIR instruction sets the working directory for any subsequent RUN, CMD, ENTRYPOINT, COPY, or ADD instruction. If the WORKDIR does not exist, it will be created even if it is not used in any subsequent Dockerfile instruction. This instruction is a readability feature to help you reduce the amount of moving around within the container.

```
» cat Dockerfile
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 .
```

**ENTRYPOINT** The ENTRYPOINT instruction allows you to configure what process is executed when a container is run.

```
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 accompanies the ENTRYPOINT and is the default arguments that will be passed to the ENTRYPOINT.

```
> docker build -t fan-club .
> docker run --rm fan-club
```

example output:

```
Richard Thomas Fan Club!
```

**CMD** CMD provides defaults for an executing container.

```
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"]
```

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

```
> docker run --rm fan-club --version
```

```
Fan Club Runner 1.0.0
```

#### Info

Occasionally you will see the CMD instruction used without the ENTRYPOINT instruction. This is only possible because the default ENTRYPOINT is `/bin/sh -c`. So `docker run -rm -it ubuntu /bin/bash` actually runs the process `/bin/sh -c /bin/bash`.

This concludes the Dockerfile instructions required for this practical, however, there are more that you may need to explore in the future. The full list of instructions can be found in the official documentation: <https://docs.docker.com/engine/reference/builder/>

### 3.1.2 Dockerfile Layers

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

Let's take a look at the Dockerfile below. In this Dockerfile, our application now has Python dependencies that need to be installed.

```
# cat Dockerfile
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 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.

```
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 real time saver, so it is worth getting into the habit early.

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
```

## Info

Note how the Ubuntu image cheats and copies the whole file structure in one layer rather than running commands to build it incrementally.

This is a common tactic in core images that the developers expect others to build on top of. One reason being the 127 layer limit of Docker.

For an example of a container that has left their whole thought process in the layers they produce, try running the Doom image.

```
$ docker run -it --rm -p 224:6901 -e VNC_PW=password kasmweb/doom:1.12.0
```

You can then go to <http://localhost:224> and connect to the VNC server with the username `kasm_user` password `password`.

```
$ docker history kasmweb/doom:1.12.0
```

### 3.1.3 Image Registries

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

The Ubuntu image that we have been using is available at [https://hub.docker.com/\\_/ubuntu](https://hub.docker.com/_/ubuntu). When browsing this page we can also 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 1: Ubuntu image on DockerHub



## 3.2 Docker Containers

Docker containers are the runtime instantiation of a Docker image. They are the process that is running inside the filesystem environment that has been created as an image.

Once you have a suitable Docker image, you can start and run a Docker container by using the `docker run` command.

## 3.3 Running a Docker Container

The below shows how to run the `hello-world` Docker image from DockerHub.

```
> docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

It is worth noting that if you do not use the `--rm` flag when using `Docker run`, the container will still exist after it has finished running. You can view the running containers with:

```
$ docker ps
```

You can view the running and stopped containers with:

```
$ docker ps -a
```

We can modify our original command so that the container is not kept around after it terminates as shown below:

```
$ docker run --rm hello-world
```

Another commonly useful set of arguments is `-i` and `-t` which combined allow you to interact with the running container process using your terminal. If we want to enter a Ubuntu environment, then we would use the combined flags, `-it` to do so.

```
$ docker run --rm -it ubuntu
```

Next we will look at allowing the outside world into our isolated environment. Remember that containers live in an isolated networking space and filesystem, however we often want to access files from our container or talk to processes running on the network inside the container.

### 3.3.1 Docker Networking

To allow our Docker containers to communicate via a network we can use the `-p` flag to specify ports we wish to publish. For example, if we are running an application in our container on port 6400, we can make it accessible outside of the container with:

```
$ docker run -p 6400:6400 fan-club
```

Note that we've repeated the port. The first occurrence of the port is the binding on the host machine, so from the host we can go to `localhost:6400` to access it. The second occurrence is the port being listened to within the container. This means that we can use different ports on the host. If we want to go to port 80 to access our service, we can do so without changing the port inside of the container with:

```
$ docker run -p 80:6400 fan-club
```

Docker networking is much more complicated than just this but it is a starting point so that we can start using containers that talk to the outside world.

### 3.3.2 Docker Volumes

The next tool to escape from a container is to use Docker volumes. Docker volumes allow a way for us to share the same folder between the host machine and the container. This is particularly useful for development of an application where the code we want to run is updated often. It is also useful if we are running an application which we would like to be persistent, like a database.

In our fan club example, we are storing our application within the `/app` directory of the container. If we want this to live update with the contents of our application on the host (assuming we run this command from the applications directory), we can do so with:

```
$ docker run -v $(pwd):/app fan-club
```

This command mounts our current working directory (`$(pwd)`) to the `/app` directory inside the container. This allows us to update our application and have the change reflected inside the container.

Again, Docker volumes are a much more complex beast but host mounting is all we require for this practical.

## 4 Containerising our Todo Application

### 4.1 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.

## 4.2 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, we will use the following Dockerfile:

```
» cat Dockerfile
FROM ubuntu:latest

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

The above image is what your local environment 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 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.

```
# Setting the working directory
WORKDIR /app

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

Note that we are installing dependencies in a layer separate to our application. This allows us to quickly rebuild the image if we change anything other than the Pipfile or Pipfile.lock.

Now that we have installed our dependencies we can copy our application into the container.

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

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

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

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

```

FROM ubuntu:latest
# Installing dependencies for running a python application
RUN apt-get update && apt-get install -y python3 python3-pip postgresql-client
    libpq-dev
# Install pipenv
RUN pip3 install pipenv
# Setting the working directory
WORKDIR /app
# Install pipenv dependencies
COPY Pipfile Pipfile.lock ./
RUN pipenv install --system --deploy
# Copying our application into the container
COPY todo todo
# Running our application
CMD ["flask", "--app", "todo", "run", "--host", "0.0.0.0", "--port", "6400"]

```

### 4.3 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 .
Successfully tagged todo:latest

```

We can then inspect our layers with the following command:

```

> docker image history todo

```

### 4.4 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 -it --rm -p 6400:6400 todo
* 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 need to make sure that you are not running the webserver from the previous practical on port 6400. If you are then 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.

## 4.5 Optimizing Our Container

We build this container image up from the Ubuntu base image. However, we rarely want to do it this way as there exists a huge range of base images for most languages and environments. Try to optimize your image by using `python:3.10` as your base image.

## 5 Docker Compose

Docker is a useful tool itself but in terms of local development, but it is not always as easy to work with as it could be. Docker Compose allows us to run multiple containers together which can be extremely useful for managing complex projects that require numerous run parameters or manage multiple containers.

For this practical, we want to use a database closer to our production environment. We will be using a PostgreSQL database which will eventually be a managed database hosted on AWS.

### 5.1 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/db:/var/lib/postgresql/data
  app:
    build: .
    restart: always
    environment:
      SQLALCHEMY_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 within the root directory.

```
$ docker-compose up
```

You should observe the output of this command and you can see two containers running. One is the database, database, and the other is our application, app. 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 change the CMD instruction to the following line in our Dockerfile:

```
# Adding a delay to our application startup
CMD ["bash", "-c", "sleep 10 && 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.

## 6 Conclusion

Over the past couple of weeks we have built a full todo app API that is ready to be deployed to a remote production 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.

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 prepare a Flask application for production as it may help you in your future projects if you choose to use Flask.

## 7 Extra

### 7.1 Python Based Dockerfile

Earlier we mentioned that you can use language specific base images for your applications. Below is an example Dockerfile that uses the python 3.10 base image.

```
FROM python:3.10
# Install pipenv
RUN pip3 install pipenv
# Setting the working directory
WORKDIR /app
# Install pipenv dependencies
COPY Pipfile Pipfile.lock ./
RUN pipenv install --system --deploy
# Copying our application into the container
COPY todo todo
# Running our application
CMD ["flask", "--app", "todo", "run", "--host", "0.0.0.0", "--port", "6400"]
```

You will notice that we did not have to install python3 and pip and the postgresql dependencies.