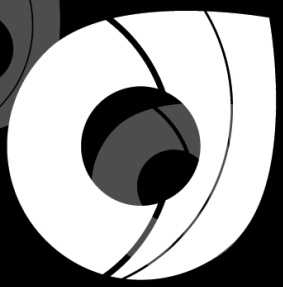
Corndel DevOps Engineering Programme

in association with Softwire

Module 5: Project Exercise



Corndel Digital.

DA97

Module 5

Project Exercise Brief

In this exercise, you will containerise your To-Do app using Docker. You'll create separate Docker images to develop, test and deploy a production-ready version of the app. We'll learn about writing custom Dockerfiles, multi-stage docker builds, configuration management and build optimisation.

Setup

Step 1: Checkout your current code

Check out your code from the previous exercise, it'll form the starting point for this exercise.

Step 2: Install Docker

If you haven't already, you'll need to install Docker Desktop. Installation instructions for Windows can be found here. If prompted to choose between using Linux or Windows containers during setup, make sure you choose Linux containers.

Submitting your work

We'll be using Pull Requests on GitHub in the same fashion as the last module to review this exercise submission.

If you experience issues, ask a tutor for help!

Exercise

Part 1: Create a production-ready container image

The primary goal of this exercise is to produce a Docker image that can be used to create containers that run the To-Do app in a production environment. As in the previous Vagrant exercise, you'll use a production-grade WSGI server to run our app in this scenario. We'll be using Gunicorn again.

Create a new file (called `Dockerfile`) in the root of your code repository. We'll include all the necessary Docker configuration in here. You can read more about dockerfile syntax here.

Create a minimal Dockerfile

The first step in creating a docker image is choosing a base image. We'll pick one from Docker Hub. A careful choice of base image can save you a lot of difficulty later, by providing many of your dependencies out-of-the-box. In this case, select one of the official Python images. Available tags combine different operating systems (e.g. `buster`, `alpine`) with different Python versions. Select one that meets your Python version requirements. The operating system is less important: `buster` or `slim-buster` (Debian 10) will be fine, and most familiar.

When complete, you should have a single line in your Dockerfile:

```
FROM <base_image_tag>
```

You can build and run your Docker image with the following commands, although it won't do anything yet!

```
$ docker build --tag todo-app .
$ docker run todo-app
```

Basic application installation

Expand the Dockerfile to include steps to import your application and launch it. You'll need to:

- Install poetry
- Copy across your application code
- Define an entrypoint, and default launch command

Keep in mind a couple Docker best practices:

- Perform the "least changing" steps early, to fully take advantage of Docker's layer caching.
- Use `COPY` to move files into your image. Don't copy unnecessary files.
- Use `RUN` to execute shell commands as part of the build process.
- `ENTRYPOINT` and `CMD` define how your container will launch.
- Add an `EXPOSE` instruction to document which port your application is listening on.

Make sure your app is running with `gunicorn`, rather than using `flask run`. *Flask's development server is not suitable for production use*. For now, continue to load your configuration values from the `.env` file - we'll come back to this later. With these steps in place, you should be able to run your app in Docker.

Rebuild your image, and re-run it. You'll need to publish the relevant port using the `-p` option. *View the app in your browser, and check it works!*

By default, docker attaches your current terminal to the container. The container will stop when you disconnect. If you want to launch your container in the background, use `docker run -d` to detach from the container. You can still view container logs using the `docker logs` command if you know the container's name or ID (if not, use `docker ls` to find the container first).

Handle secret values

There is a potential security issue with your approach so far. The `.env` file contains application secrets (API keys), and it is included in the docker image. This is bad practice. Anyone with access to the image (which you may make public) can discover the embedded content.

It's good practice for containerised applications to be configured *only* via environment variables, as they are a standard, cross-platform solution to configuration management. Instead of copying in a configuration file (`.env`) at build-time, we need to pass docker the relevant environment variables at runtime. This will keep your secrets safe, while also keeping your image re-usable - you could spin up

multiple containers, each using different credentials.

First, change your docker run command to load environment variables at runtime using the `--env-file` option. This can be done as part of the `docker run` command. That way, your Dockerfile won't need to know anything about `.env`.

Second, create a `.dockerignore` file, and use to it specify files and directories that should never be copied to docker images. This can include things like your vagrant configuration, secrets (`.env` and `.env.template`) and unwanted IDE or tool configuration directories (e.g. `.git`). Anything that will never be required to run or test your application should be registered with `.dockerignore` to improve your build speed and reduce the size of the resulting images.

Note that any environment variables loaded as part of `docker run` will overwrite any defined within the Dockerfile using the `ENV`.

Part 2: Create a local development container

Containers are not only useful for production deployment. They can encapsulate the programming languages, libraries and runtimes needed to develop a project, and keep those dependencies separate from the rest of your system. In the previous project exercise you used Vagrant to create a portable local development environment. Here you'll do something similar with Docker.

In Part 1 you created what's known as a "single-stage" docker image. It starts from a base image, adds some new layers and produces a new image that you can run. The resulting image can run the app in a production manner, but is not ideal for local development. Your local development image should have two key behaviours:

- Enable Flask's debugging/developer mode to provide detailed logging and feedback.
- Allow rapid changes to code files without having to rebuild the image each time.

To do this, you will convert your Dockerfile into a "multi-stage" docker file. Multi-stage builds can be used to generate different variants of a container (e.g. a development container, a testing container and a production container) from the same

Dockerfile. You can read more about the technique here.

Here is an outline for a multi-stage build:

```
FROM <base-python-image> as base

# Perform common operations, dependency installation etc...

FROM base as production

# Configure for production

FROM base as development

# Configure for local development
```

Since Flask's development server allows hot reloading of code changes while running, we'll use that instead of Gunicorn to run the app in our development container. The only thing we need to consider in order to achieve the second requirement is how to make changes to the code files and have them appear within the container without having to rebuild the image each time we modify the code. The solution is to use a bind mount when running the container to make the project directory on your host machine available as a mounted directory within the container.

The goal is to be able to create either a development or production image from the same Dockerfile, using commands like:

```
$ docker build --target development --tag todo-app:dev .
$ docker build --target production --tag todo-app:prod .
```

On UNIX shells, you can test the local development setup using a command similar to:

```
$ docker run --env-file ./.env -p 5100:80 --mount type=bind,source="$(pwd)"/
todo_app,target=/app/todo_app todo-app:dev
```

You can also use the `-d` flag to detach from the docker shell. This will launch your container in the background. To view the container logs, you'll need to use `docker logs <CONTAINER>`

Part 3: Optimise your Dockerfile

Have a look at your Dockerfile again. Is there any way you could optimise it? Docker optimisation has two main considerations: speed and size. Optimising image size is challenging with an interpreted language like Python, as the application requires a Python interpreter, complete with standard library. You can, however, ensure that you only import what needed to run the application rather than the entire git repository. You can view the size of your docker images using `docker ls`.

For this part, we'll focus on build speed optimisation.

Docker caches every layer it creates, making subsequent re-builds extremely fast. But that only works if the layers don't change. For example, Docker should not need to re-install your project dependencies because you apply a small bug fix to your application code.

Docker must rebuild a layer if:

1. The command in the dockerfile changes
2. Files referenced by a `COPY` or `ADD` command are changed.
3. Any previous layer in the image is rebuilt.

You should place largely unchanging steps towards the top of your Dockerfile (e.g. installing build tools), and apply the move frequently changing steps towards the end (e.g. copying application code to the container).

With this in mind, review your Dockerfile. Is there any way you could improve it?

Part 4 (Stretch Goal): Use Docker Compose

Launching containers with long `docker run` commands can become tedious, and difficult to share with other developers. Here we'll introduce **docker compose**, a tool bundled with docker that can automate the launch, networking, and lifecycle management of, containers. You can read more about it in the docs. The basic principle is that you write all the configuration necessary to launch your containers in a single yaml file (`docker-compose.yml`), and a single command (`docker-compose up`) can then be used to launch them.

- Create a `docker-compose.yml` that launches and persists your development container.
- Use `docker-compose up` to launch your new, containerised local development environment.

Part 5 (Stretch Goal): Run your tests in Docker

- Add a third build stage that encapsulates a complete test environment. Use the image to run your unit, integration and end-to-end tests with `docker run`.
- Modify your test container to be persistent, and re-run tests whenever it detects a change to the source files. (hint: use a bind mount, and a command line tool that can watch for file system events e.g. watchdog).
- Expand your `docker-compose.yml` file to include your persistent test container(s). `docker-compose up` should now launch your local development environment and persistent test runners.

Hints

1. By default, docker runs commands inside the container as `root`. This means you should not need to worry about file permissions, nor execute any commands with `sudo`.
2. You can use the `RUN` instruction to your Dockerfile to execute arbitrary shell commands (for example, to install poetry). By default, `RUN` uses the Bourne shell (`sh`), which may lack some features you want. If that's the case, use the *exec* form to specify a different shell.
3. Don't try to modify your shell environment as part of `RUN`. The changes won't persist to layers beyond the current `RUN` instruction. Instead, use Docker's `ENV` instruction to set persistent environment variables at build time.
4. The `COPY` instruction can load data from the docker build context into the docker image.
5. If Docker can't find `poetry`, are you sure you've installed it? You may need to modify the `PATH` environment variable.
6. If Docker can't find your app code, are you sure you've copied it to the image? You may need to modify the `PYTHONPATH` environment variable.
7. Think about how to install your dependencies using poetry. In particular, the `--no-root` and `-no-dev` flags may be of interest.