Lab 8.2: Docker Review
ID720 Virtualisation

Introduction

When I started organising this paper several years ago, containers and Docker were pretty cutting edge. Now they're pretty routine and most of you are probably already familiar with these things. Next week we will start looking at some more advanced topics, but we will start with a quick review of things with which you should be familiar.

Today's lab should be carried out on our OpenStack VM that we used over the last few weeks. You may be able to do this lab on your own machine, but it would require some setup that we will not cover today.

1 Working with Docker

We interact with Docker primarily with the 'docker' command. To start, use the `docker info` command to verify that Docker is installed and that you can use it.

Now we can run an actual container. Create a new container with the command

```
  docker run -i -t --name <your username> ubuntu /bin/bash
```

This will create a new container named with your username based on the ubuntu base image. We have told docker to run bash on the container, and the -i and -t options connect us to an interactive console on it.

Once the container is up you can interact with it normally. A few things about your container environment are interesting. Run `top` to see what is running inside the container. For comparison, you may want to run `top` on the host system when you exit the container. Also, use `ip a` to inspect the container?s network interfaces.

Type `exit` to return to the host. Since this terminates the bash shell the container itself stops.

On the host system, type `docker ps -a` to list the containers on the system. (Without the `-a` it will only show running containers)

You can get more information about your container with the command

```
docker inspect <your username>
```

Now restart your container with the command

```
docker start <your username>
```

and run `docker ps`. You will see that the container is running, but we are not attached to the console. You can attach to it with the command

```
docker attach <your username>
```

2 Building an image

Now we can build our own images. To do this we'll need to create a working directory that we call a *build context*. We organise our image's resources in this directory. Create a directory called `lab8.2` for this.

Inside your new build context directory create a new file called `Dockerfile`. We will edit this file, adding a series of directives, one per line, that build up our container in a step-by-step manner.

Add the following lines to your Dockerfile:

```
FROM ubuntu:20.04
```

Dockerfiles always start with a `FROM` statement that specifies the base image type from which this container is built.

Your next line should be

```
LABEL org.opencontainers.image.authors="<your email address>"
```

This identifies the person responsible for this image.

```
RUN apt-get -q update && apt-get -yq dist-upgrade
RUN apt-get -yq install apache2
```

`RUN` commands run their arguments as if they were entered into a shell inside the container.

```
ENV DEBIAN_FRONTEND=noninteractive
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_LOCK_DIR /var/run/apache
ENV APACHE_PID_FILE /var/run/apache/httpd.pid
ENV APACHE_RUN_DIR /var/run/apache2
RUN mkdir /var/run/apache2
```

`ENV` commands populate environment variables inside the container with values. This is a common way to pass configuration information to services running inside them.

```
ADD index.html /var/www/html/index.html
```

This command will take a file named `index.html` from your build context and place a copy at the location specified inside the container. If the first argument to `ADD` is a directory, indicated by a trailing slash, then it will add the directory and its contents.

```
EXPOSE 80
```

This will cause our container to open port 80 on the host system. Note that this doesn't mean that the host's port 80, or any other port, will be open or associated with the container's port. This is done with the `docker run` comand.

```
ENTRYPOINT ["/usr/sbin/apache2"]
CMD["-DFOREGROUND"]
```

Recall that Docker containers typically run one main process. The `ENTRYPOINT` specifies the command to run inside the container to start that process. The following `CMD` provides arguments to the `ENTRYPOINT`'s command.

This complete's our Dockerfile. Since the `ADD` command specified an HTML file, you'll need to place one inside your build context alongside the Dockerfile. This can just be a simple hello world page.

Don't forget to commit and push your changes.

From within your build context directory, build your new image with the command

```
docker build -t="your-username/lab8.2" .
```

3 Run a container

Once this completes successfully, you can run a container based on this image with the command

```
docker run –name <your-username-lab8> -p 127.0.0.1:80:80XX your-username/lab8.2
```

The `-p` option maps the exposed port `80` on our container to port `80XX` on the loopback interface. Since we are sharing a host, select two random numbers to substitute for the "`XX`" that do not clash with other students' ports. You can see your web page with the command

```
curl http://localhost:80XX/
```

Using your chosen port number.

If you make any changes to your build context you will need to rebuild the image, but this will not affect any containers built from older versions of the image.

4 More information

There are many more dockerfile directives beyond the ones we used here. You can consult the text to learn more, and you can check online documentation at `https://docs.docker.com/engine/reference/builder/`. Note that the web site will have somewhat more up-to-date information.