Lab 11.1: Docker Networking
IN720 Virtualisation

Introduction

So far we have mainly worked with just one container at a time, and we generally stick to the guideline that a container runs just one main process. Most real services are provided by multiple processes working in concert, which means that we expect to deploy more than one container to provide the service. In order to get those containers to work together, we need to use Docker networking.

We will start this lab with a demonstration of the properties of bridged networking with Docker. Then we will create a simple service using a Python/Flask application in one container and an nginx reverse proxy server in a second container. We will create a Docker network to connect the two containers.

1 Examine Docker networks

First, see what networks are currently present on your system with the command `docker network ls`. If you haven't done anything with Docker networks on this host yet, you should see three networks: the default bridged network, the host network, and the "none" network.

Create an additional bridged network with the command

```
docker network create --driver bridge lab-net
```

Since the `bridge` network driver is the default it's not really necessary to do so, but we include it here for clarity.

Next, let's launch some containers. First, two containers using the default network:

```
docker run -dit --name lab11a ubuntu /bin/bash
docker run -dit --name lab11b ubuntu /bin/bash
```

We used the `-d` option, so we have to explicitly attach to the containers if we want to work in the shells we launched.

Launch two more containers, but this time place them on the `lab-net` we defined earlier:

```
docker run -dit --name lab11c --network lab-net ubuntu /bin/bash
docker run -dit --name lab11d --network lab-net ubuntu /bin/bash
```

Now examine both networks with the commands `docker network inspect bridge` and `docker network inspect lab-net`. You should see our four containers attached to their respective networks. Note their ip addresses.

Attach to the `lab11a` container. Install `ping` with the command

```
apt update && apt install iputils-ping
```

Then try pinging other containers to see what happens:

```
ping <ip address of lab11b>
ping <ip address of lab11c>
ping lab11b
```

Detach from the container's without exiting the shell by entering `Ctrl-p Ctrl-q`.

Now, contrast this by attaching to `lab11c`, installing ping, and trying

```
ping <ip address of lab11d>
ping <ip address of lab11b>
ping lab11d
```

The results show us two things. First, that the respective networks are isolated from each other. Second, that on user-defined networks we can connect to other containers by name. This proves useful, as we'll see in the following task.

2 Create a Flask application container

Make a directory `flaskapp` to serve as a build context for the Python/Flask application image. In that directory, create a dockerfile with the following contents:

```
FROM ubuntu:16.04
LABEL updated_on="2019-10-18 09:00"

RUN apt-get update
RUN apt-get -y upgrade
RUN apt-get -y install python3 python3-setuptools python3-pip gunicorn3
RUN  update-alternatives --install /usr/bin/python python /usr/bin/python3 10

COPY lab-4-app /flaskapp
WORKDIR /flaskapp
RUN pip3 install -r requirements.txt

EXPOSE 5000
ENTRYPOINT "./startup.sh"
```

This dockerfile creates an image with some dependencies needed to run our app, copies our Flask application code into it, and then runs the app using a gunicorn wsgi server on port 5000. Now we just need to add the application code to our context. Clone the GitHub repository tclark/lab-4-app[1] into your context.

Build your container image with the tag `user/flaskapp`. Test it by running a container with the following command

```
docker run -d --rm --name flaskapp -p 5000:5000 user/flaskapp
```

and verify that it works by checking it with a browser. Our application works, but for a production deployment we need to place it behind a reverse proxy server. For this, we will need a second container. Shut down your `flaskapp` container.

3 Create an nginx Container

Set up a second build context, named `nginx`. Place a dockerfile in it with the following:

```
FROM nginx:1.13
COPY flaskapp.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
```

[1] This lab used to take place in week 4 and I didn't rename the Git repo.

This will produce a simple nginx container to serve as a front end for our service. We need to add the `flaskapp.conf` file to our context with the following:

```
resolver 127.0.0.11 valid=1s;

server {
  set $alias "flaskapp";
  location / {
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_pass http://$alias:5000;
  }
  listen 80;
}
```

This will cause nginx to send proxy requests to a `flaskapp` host, or in our case, container. Build this image with the tag `user/nginx`.

4 Test our service

Now, run two containers based on our images with the following commands.

```
docker run -d --rm --name flaskapp user/flaskapp
docker run -d --rm --name nginx -p 8080:80 user/nginx
```

and see if you can see your application output.

Stop your containers.

5 Create a Docker Network

We want to place our containers on their own isolated network. The Flask application container does not need to be reachable by anything but our nginx container. Create our network with the command

```
docker network create app
```

Now we just need to start our containers again, this time directing them to use this new network.

```
docker run -d --rm --name flaskapp --network app user/flaskapp
docker run -d --rm --name nginx --network app -p 8080:80 user/nginx
```

Note that it doesn't matter in which order you start the containers. Now verify one last time that your application works by checking it with a browser.