QA Consulting

Docker Networking

TRAINING MATERIALS - MODULE HANDOUT

Contacts

robert.crutchley@qa.com
team.qac.all.trainers@qa.com

www.consulting.qa.com

Contents

Overview

We can utilise networking in Docker when we would like to have multiple containers working together. For example a frontend and backend application with a database. The frontend application needs to be able to communicate with the backend and the backend application needs to be able to communicate with the database.

We can have these three different applications running in separate containers. This is because if they were all in the same container, then everything would need to be redeployed whenever any of the three have been updated.

Networks in Docker

There are several different types of network drivers, for different networking purposes. Some are for single host solutions while others are for spreading containers across more than one host. The main focus for us will be with bridge networks because overlay networks are for Docker Swarm setups and macvlan networks are for legacy applications.

Bridged Networks

Bridged networks are for connecting multiple containers on a single host. You will be able to access the container with it's private IP address. Mapping ports to the host needs to be done explicitly.

Host Networks

These should be used when you want container application ports to map to the host automatically. Instead of explicitly publishing a port from a container to the host, applications that listen on a port will be available from the host on that same port. Be care of ports clashing between applications when you use this.

Overlay Networks

This is where Docker allows you to scale your application, overlay networks allows network connectivity over multiple hosts. Your applications deployed on different hosts will be able to communicate to each other, this kind of networking appears when using Docker Swarm, a container orchestration tool but can also be used for standalone containers.

Macvlan Networks

Some legacy applications or tools which monitor network traffic expect to be on a physical network, which Docker networks are definitely not. Macvlan networks are for these kinds of applications.

Managing Docker Networks

Connecting a New Container to a Network

Containers can set to connect to a network when you are creating them by providing the **--network** flag.

```
docker network create [NETWORK_NAME]
```

```
docker network create my-network
```

Connect an Already Existing Container to a Network

If you have a container that exists already but you want to add it to a network without having to recreate it then you can use the **network connect** command, providing the network name and the container name.

```
docker network connect [NETWORK_NAME] [CONTAINER_NAME]
```

```
docker network connect my-network my-container
```

Disconnect a Container from a Network

Disconnecting a container from a network can be done just like when you connect an existing container to a network, provide the network name and the container name.

```
docker network disconnect [NETWORK_NAME] [CONTAINER_NAME]
```

```
docker network disconnect my-network my-container
```

Delete a Network

You can only delete a network if there are not containers connected to it otherwise you will get an error message.

```
docker network rm [NETWORK_NAME]
```

```
docker network rm my-network
```

Viewing your Docker Networks

You can view what networks you already have along with their details such as what type of network they are and the name of them.

```
docker network list
```

Bridged Networks

Overview

There are two kinds of bridged networks, user defined and default. When containers are started a **--link** flag can be provided to link it to another container using the default bridge network. The default bridge is legacy method in Docker for networking and is not recommended anymore by Docker, so we shall avoid that going forward.

User defined bridged networks provide many benefits

- **Isolation from other containers on the same host that aren't in the same network.**
  When two containers are on the same user defined bridge network, all ports are open to each other and they can communicate freely.
- **Automatic DNS Resolution**
  When a container is put on a bridge network it will have a private IP address. A DNS entry is automatically configured to this IP address which is just the container name. For instance if there were two containers running, one called **nginx** and the other called **application**, and the application container had a server listening on port 8080. The nginx container would be able to connect to the application running in the other container by its name (**http://application:8080**).

- **Connect and Disconnect Containers on the Fly**
  You can connect and disconnect containers to a bridge network without having to stop the containers.

Create Bridged Networks

When you create a network with the **docker network** command the default network type is bridged. To create a bridged network you need to provide the name of the new network.

```
docker network create [NETWORK_NAME]
```

```
docker network create my-network
```

Host Networks

Something to keep in mind with host networks is at the time of writing this host networks are only available on Linux hosts running Docker. If you list the networks that you have you should see that there is one network called host. To get a container onto this network you can just specify the network name when connecting the container to it.

Tasks

This exercise is for demonstrating how bridge and host network drivers in Docker behave. For the host driver we will see how we can access our deployed application on the host without needing to publish any ports. With the bridge driver we will configure an NGINX reverse proxy for an application, which will require two containers to communicate with each other.

Create a new folder (~/docker/networking_exercise) for this exercise and make sure that you are running your commands and creating the required files in this folder.

Creating a Container on the Host Network Driver

We'll deploy the default NGINX image and attach the container to the host network. Once the container has started you will be able to access it in your browser by using a curl command on localhost.

```
docker run -d --network host --name nginx nginx
```

Stop and remove the nginx container when you are done.

Create a New Bridged Network

We'll create a new bridged network for our reverse proxy setup.

```
docker network create my-network
```

Create an Application Container

For this example we can use Jenkins which is a very popular CI tool that is web based, you may deploy your own application if you wish though. Create the application container and attach it to the new bridge network.

```
docker run -d --network my-network --name jenkins jenkins
```

Create an NGINX Configuration

NGINX is another very popular tool that can address many networking issues, we'll be using it for it's primary purpose here, as a reverse proxy. We'll need create our own config file for this (~/docker/networking_exercise/nginx.conf).

nginx.conf

```
events {}
http {
    server {
        listen 80;
        location / {
            proxy_pass http://jenkins:8080;
        }
    }
}
```

Create an NGINX Container Using Our Configurations

We are going to pass in the configuration that we made as volume to the container, this handout doesn't explain what this is however all you need to know at this point is that it gives the container access to the configuration file that we just created. The NGINX container that we create must also be attached to the network we made earlier We'll also publish port 80 we can access NGINX from outside of the bridge network.

```
docker run -d --network my-network -v
$(pwd)/nginx.conf:/etc/nginx/nginx.conf -p 80:80 --name nginx nginx
```

Access Your Application

You should now be able to access your application from the host, http://localhost, or if it a remote server with port 80 opened on the firewall you can just access your application via the public IP address.

When you connect to your application the traffic is going through NGINX first, it is then routed to you deployed application even though they are in separate containers. This made possible because both the containers are on the same bridge network.

Try it on our Own Projects

Stop and remove all the containers, delete the bridge network that was created.
Can you get this working on one of your own projects? For instance a server that connects to a database, or a frontend application that connects to a server using Docker networks.