Docker Networking Basics

Objective: In this lesson we will begin discussing the basics of networking with Docker. We will look at exploring networking options and how to network containers together.<br>
Preparation: Open up two terminal windows to serve as a primary and secondary. Also, cd into the appropriate lab file, before begining.<br>
Outcome: Once complete, the participant will have a created network and connected containers to it. Also, link a container to another, mount volumes during launch, and by way of a Dockerfile. <br>
Data Files: All files located in the “lab03:Material” folder.<br>

Step 1. Using a Docker Container Network

1. Creating a Docker network is easy. Let’s start by familiarizing ourselves with Docker network syntax and options by using our help command, as follows:

```
$ docker create --help
```

Output:

```
Usage:  docker network COMMAND
```

Manage networks:

```
Options:
  --help, -h   Print usage
Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks
```

The first real step is to create a network using the CLI. This network will allow us to attach multiple containers which will be able to discover each other.

In this example, we’re going to start by creating a `backend-network`. All containers attached to our backend will be on this network.

2. Create Network-To start with we create the network with our predefined name.

```
docker network create backend-network
```

3. Connect To Network-When we launch new containers, we can use the `--net` attribute to assign which network they should be connected to.

```
docker run -d --name=redis --net=backend-network redis
```

In the next step we'll explore the state of the network.

Unlike using links, `docker network` behaves like traditional networks where nodes can be attached/detached.

4. Explore-The first thing you'll notice is that Docker no longer assigns environment variables or updates the hosts file of containers. Explore using the following two commands and you'll notice it no longer mentions other containers.

```
docker run --net=backend-network alpine env
docker run --net=backend-network alpine cat /etc/hosts
```

5. Instead, the way containers can communicate via an Embedded DNS Server in Docker. This DNS server is assigned to all containers via the IP `127.0.0.11` and set in the `resolv.conf` file.

```
docker run --net=backend-network alpine cat /etc/resolv.c
onf
```

6. When containers attempt to access other containers via a well-known name, such as Redis, the DNS server will return the IP address of the correct Container. In this case, the fully qualified name of Redis will be `redis.backend-network`.

```
docker run --net=backend-network alpine ping -c1 redis
```

Docker supports multiple networks and containers being attached to more than one network at a time.

7. For example, let’s create a separate network with a Node.js application that communicates with our existing Redis instance.The first task is to create a new network in the same way.

```
docker network create frontend-network
```

8. When using the connect command it is possible to attach existing containers to the network.

```
docker network connect frontend-network redis
```

9. When we launch the web server, given it’s attached to the same network it will be able to communicate with our Redis instance.

```
docker run -d -p 3000:3000 --net=frontend-network katacod
a/redis-node-docker-example
```

10. You can test it using `curl <hostname>:3000`

Links are still supported when using docker network and provide a way to define an Alias to the container name. This will give the container an extra DNS entry name and way to be discovered. When using `--link` the embedded DNS will guarantee that localised lookup result only on that container where the `--link` is used.

The other approach is to provide an alias when connecting a container to a network.

11. Connect Container with Alias-The following command will connect our Redis instance to the frontend-network with the alias of db.

```
docker network create frontend-network2
docker network connect --alias db frontend-network2 redis
```

12. When containers attempt to access a service via the name db, they will be given the IP address of our Redis container.

```
docker run --net=frontend-network2 alpine ping -c1 db
```

With our networks created, we can use the CLI to explore the details.

13. The following command will list all the networks on our host.

```
docker network ls
```

14. We can then explore the network to see which containers are attached and their IP addresses.

```
docker network inspect frontend-network
```

15. The following command disconnects the redis container from the frontend-network.

```
docker network disconnect frontend-network redis
```

Step 2. Exposing Docker Container Ports

1. Create an empty directory and copy the follwoing code in it, with the name Dockerfile. Take note of the comments that explain each statement. Notice, this is where port mapping begins with exposing port 80 in our Dockerfile.

```
# Use an official Python runtime as a base image
FROM python:2.7-slim


# Set the working directory to /app
WORKDIR /app


# Copy the current directory contents into the container
at /app
ADD . /app


# Install any needed packages specified in requirements.t
xt
RUN pip install -r requirements.txt


# Make port 80 available to the world outside this contai
ner
```

```
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

This Dockerfile refers to a couple of things we haven't discussed yet, but don't worry. Just move along and we will get there.

2. In the Dockerfile, a few things are mentioned that we haven't created. Confirm that your Dockerfile is in the same folder as the `app.py` and `requirements.txt` files mentioned.

```
$ ls
```

Output:

```
Dockerfile      app.py          requirements.txt        ..
.
```

> *NOTE: There might be more folders then this, but confirm you have at least these three. If these three are not in the same folder this app won't work.*

3. These two files work with our Dockerfile to make this app work.

We are going to launch the app in a moment, but first take a moment to view the contents of these two files.

```
$ cat app.py
$ cat requirements.txt
```

Now we see that `pip install -r requirements.txt` installs the Flask and Redis libraries for Python, and the app prints the environment variable `NAME`, as well as the output of a call to `socket.gethostname()`. Finally, because Redis isn't running (as we've only installed the Python library, and not Redis itself), we should expect that the attempt to use it here will fail and produce the error message.

> *NOTE: Accessing the name of the host when inside a container retrieves the container ID, which is like the process ID for a running executable.*

4. Now run the build command. This creates a Docker image, which we're going to tag using -t so it has a friendly name.

```
$ docker build -t friendlyhello .
```

5. Where is your built image? It's in your machine's local Docker image registry:

```
$ docker images
```

Output:

```
REPOSITORY            TAG                 IMAGE ID
friendlyhello         latest              326387cea398
```

6. Where is your built image? It's in your machine's local Docker image registry:

```
$ docker images
```

Output:

```
REPOSITORY            TAG                 IMAGE ID
friendlyhello         latest              326387cea398
```

7. Run the app, mapping your machine's port `4000` to the container's exposed port `80` using `-p`:

```
docker run -p 4000:80 friendlyhello
```

You should see a notice that Python is serving your app at `http://0.0.0.0:80`. But that message is coming from inside the container, which doesn't know you mapped port `80` of that container to `4000`, making the correct URL `http://localhost:4000`.

8. Go to that URL in a web browser to see the display content served up on a web page, including "Hello World" text, the

container ID, and the Redis error message.

9. You can also use the `curl` command in a shell to view the same content.

```
$ curl http://localhost:4000
```

Output:

```
<h3>Hello World!</h3><b>Hostname:</b> 8fc990912a14<br/><b
>Visits:</b> <i>cannot connect to Redis, counter disabled
</i>
```

> *NOTE: This port remapping of `4000:80` is to demonstrate the difference between what you `EXPOSE` within the `Dockerfile`, and what you publish using `docker run -p`. In later steps, we'll just map port `80` on the host to port `80` in the container and use `http://localhost`.*

10. Hit `CTRL+C` in your terminal to quit.

11. Now let's run the app in the background, in detached mode:

```
docker run -d -p 4000:80 friendlyhello
```

12. You get the long container ID for your app and then are kicked

back to your terminal. Your container is running in the background. You can also see the abbreviated container ID with docker ps (and both work interchangeably when running commands):

```
$ docker ps
```

Output:

```
CONTAINER ID        IMAGE               COMMAND
   CREATED
1fa4ab2cf395        friendlyhello       "python app.py"
   28 seconds ago
```

You’ll see that CONTAINER ID matches what’s on `http://localhost:4000`.

13. Now use docker stop to end the process, using the CONTAINER ID, like so:

```
docker stop 1fa4ab2cf395
```

Step 3. Linking Docker Containers Directly

The most common scenario for connecting to containers is an application connecting to a data-store. The key aspect when creating a link is the name of the container. All containers have names, but to make it easier when working with links, it’s important to define a friendly name of the source container which you’re connecting too.

In this example, we bring up a Alpine container which is linked to our `redis-server`. We’ve defined the alias as `redis`. When a link is created, Docker will do two things.

1. Run a redis server with a friendly name of redis-server which we’ll connect to in the next step. This will be our source container.

```
docker run -d --name redis-server redis
```

> *NOTE: Redis is a fast, open source, key-value data store.*

To connect to a source container you use the `--link <container-name|id>:<alias>` option when launching a new container. The container name refers to the source container we defined in the previous step while the alias defines the friendly name of the host.

2. Docker will set some environment variables based on the linked to the container. These environment variables give you a way to reference information such as Ports and IP addresses via known names.You can output all the environment variables with the env command. For example:

```
docker run --link redis-server:redis alpine env
```

Output will resemble:

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/s
bin:/bin
HOSTNAME=0003b296322d
REDIS_PORT=tcp://172.18.0.2:6379
REDIS_PORT_6379_TCP=tcp://172.18.0.2:6379
REDIS_PORT_6379_TCP_ADDR=172.18.0.2
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_NAME=/pedantic_brattain/redis
REDIS_ENV_GOSU_VERSION=1.7
REDIS_ENV_REDIS_VERSION=3.2.8
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/rel
eases/redis-3.2.8.tar.gz
REDIS_ENV_REDIS_DOWNLOAD_SHA1=6780d1abb66f33a97aad0edbe02
0403d0a15b67f
HOME=/root
```

3. Docker will update the HOSTS file of the container with an entry for our source container with three names, the original, the alias and the hash-id. You can output the containers host entry using cat /etc/hosts

```
docker run --link redis-server:redis alpine cat /etc/host
s
```

Output will resemble:

```
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.18.0.2      redis c0efb9fdf4c7 redis-server
172.18.0.3      a75804ff101d
```

4. Example-With a link created you can ping the source container in the same way as if it were a server running in your network.

```
docker run --link redis-server:redis alpine ping -c 1 red
is
```

Output:

```
--- redis ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.083/0.083/0.083 ms
```

With a link created, applications can connect and communicate with the source container in the usual way, independent of the fact both services are running in containers.

5. For example, here is a simple Node.js application which connects to the Redis container, using the hostname `redis`.

```
docker run -d -p 3000:3000 --link redis-server:redis kata
coda/redis-node-docker-example
```

6. Sending an HTTP request to the application will store the request in Redis and return a count. If you issue multiple requests, you'll see the counter increment as items are persisted.

```
$ curl <hostname>:3000
```

> *NOTE: Most likely your hostname is `localhost`, but ask your instructor if you require assistence.*

7. Run `curl <hostname>:3000` a few times and enjoy watching the count climb.

In the same way, you can connect source containers to applications, you can also connect them to their own CLI tools.

8. The command below will launch an instance of the Redis-cli tool and connect to the redis server via it's alias.

```
docker run -it --link redis-server:redis redis redis-cli
-h redis
```

9. If you are familiar with Redis, feel free to climb around. The command `KEYS *` will output the contents stored currently in the source redis container.

10. Type `QUIT` to exit the CLI.

Step 4. Mounting Volumes

Docker Volumes are created and assigned when containers are started. Data Volumes allow you to map a host directory to a container for sharing data.

This mapping is bi-directional. It allows data stored on the host to be accessed from within the container. It also means data saved by the process inside the container is persisted on the host.This example will use Redis as a way to persist data.

1. Start a Redis container below, and create a data volume using the `-v` parameter. This specifies that any data saved inside the container to the `/data` directory should be persisted on the host in the directory `/docker/redis-data`.

```
docker run  -v <path-to-lab03:material>/redis-data:/data
\
  --name r1 -d redis \
  redis-server --appendonly yes
```

2. We can pipe data into the Redis instance using the following

command.

```
cat data | docker exec -i r1 redis-cli --pipe
```

3. Redis will save this data to disk. On the host we can investigate the mapped direct which should contain the Redis data file.

```
ls <path-to-lab03:material>/redis-data
```

4. This same directory can be mounted to a second container. One usage is to have a Docker Container performing backup operations on your data.

```
docker run  -v <path-to-lab03:material>/redis-data:/backu
p ubuntu ls /backup
```

Data Volumes mapped to the host are great for persisting data. However, to gain access to them from another container you need to know the exact path which can make it error-prone. An alternate approach is to use `-volumes-from`. The parameter maps the mapped volumes from the source container to the container being launched.

5. In this case, we're mapping our Redis container's volume to an Ubuntu container. The `/dat`a directory only exists within our Redis container, however, because of `-volumes-from` our Ubuntu container can access the data.

```
docker run --volumes-from r1 -it ubuntu ls /data
```

This allows us to access volumes from other containers without having to be concerned how they're persisted on the host.

Mounting Volumes gives the container full read and write access to the directory. You can specify read-only permissions on the directory by adding the permissions :ro to the mount.

6. If the container attempts to modify data within the directory it will error.

```
docker run -v <path-to-lab03:material>/redis-data:/data:r
o -it ubuntu rm -rf /data
```

Conclusion

In this lesson we covered creating networks, port mapping from the CLI and Dockerfile, linking containers, and persistent data with volume mounting. Hopesfully you are starting to get an idea of Docker's full potenial. Before moving on though, make sure you stop each container and do a general reset of your environment. Great work!