LAB 8 Docker networking and volumes

Docker networking
So far, we have been launching our containers on a single flat shared network. Although we have not talked about it yet, this means the containers we have been launching would have been able to communicate with each other without having to use any of the host networking.

Rather than going into detail now, let's work through an example. We are going to be running a two-container application; the first container will be running Redis and the second our application, which uses the Redis container to store a system state.

Redis is an in-memory data structure store that can be used as a database, cache, or message broker. It supports different levels of on-disk persistence. For more information, refer to https://redis.io/.

Before we launch our application, let's download the container images we will be using and also create the network:

$ docker image pull redis:alpine
$ docker image pull russmckendrick/moby-counter
$ docker network create moby-counter
Managing Containers

You should see something similar to the following Terminal output:

```
$ docker container run -d --name redis --network moby-counter redis:alpine
```

As you can see, we have used the `--network` flag to define which network our container was launched in. Now that the Redis container is launched, we can launch the application container by running the following:

```
$ docker container run -d --name moby-counter --network moby-counter -p 8080:80 russmckendrick/moby-counter
```
Managing Containers

Again, we have launched the container into the moby-counter network; this time, we have mapped port 8080 to port 80 on the container. Notice that we did not need to worry about exposing any ports of the Redis container. That is because the Redis image comes with some defaults, which expose the default port (which is 6379) for us. This can be seen by running docker container ls:

```
<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
<th>NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>8cd5b63dedd1</td>
<td>russmckendrick/moby-counter</td>
<td>&quot;node index.js&quot;</td>
<td>14 hours ago</td>
<td>Up 12 seconds</td>
<td>0.0.0.0:8000-&gt;80/tcp</td>
<td>moby-counter</td>
</tr>
<tr>
<td>1bde5538e5d8</td>
<td>redis:alpine</td>
<td>&quot;docker-entrypoint...&quot;</td>
<td>14 hours ago</td>
<td>Up 13 seconds</td>
<td>6379/tcp</td>
<td>redis</td>
</tr>
</tbody>
</table>
```

All that remains now is to access the application; to do this, open your browser and go to http://localhost:8080/. You should be greeted by a mostly blank page with the message Click to add logos:
Clicking anywhere on the page will add Docker logos, so click away:

So what is happening? The application that is being served from the `moby-counter` container is making a connection to the `redis` container and using the Redis service to store the on-screen coordinates of each of the logos you are placing on the screen by clicking.

How is the `moby-counter` application connecting to the `redis` container? Well, in the `server.js` file, the following default values are being set:

```javascript
var port = opts.redis_port || process.env.USE_REDIS_PORT || 6379
var host = opts.redis_host || process.env.USE_REDIS_HOST || 'redis'
```

This means that the `moby-counter` application is looking to connect to the host called `redis` on port `6379`. Let's try using the `exec` command to ping the `redis` container from the `moby-counter` application and see what we get:

```
$ docker container exec moby-counter ping -c 3 redis
```
You should see something similar to the following output:

PING redis (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.057 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.087 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.085 ms

--- redis ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.057/0.076/0.087 ms

As you can see, the moby-container container resolves redis to the IP address of the Redis container, which is 172.18.0.2. You may be thinking that the application's host file contains an entry for the redis container; let's take a look using the following command:

$ docker container exec moby-counter cat /etc/hosts

This returns the contents of /etc/hosts, which for me look like the following:

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
172.18.0.3 8ed5183eded1

Other than the entry at the end, which is actually the IP address resolving to the hostname of the local container, 8ed5183eded1 is the ID of the container; there is no sign of an entry for redis. Next, let's check /etc/resolv.conf by running the following command:

$ docker container exec moby-counter cat /etc/resolv.conf

This returns what we are looking for; as you can see, we are using a local nameserver:

nameserver 127.0.0.11
options ndots:0

Let's perform a DNS lookup on redis against 127.0.0.11 using the following command:

$ docker container exec moby-counter nslookup redis 127.0.0.11
Managing Containers

This returns the IP address of the redis container:

    Server: 127.0.0.11
    Address 1: 127.0.0.11

    Name: redis.moby-counter
    Address 1: 172.18.0.2 redis.moby-counter

Let's create a second network and launch another application container:

    $ docker network create moby-counter2
    $ docker run -itd --name moby-counter2 --network moby-counter2 -p 9090:80
      russmckendrick/moby-counter

Now that we have the second application container up and running, let's try pinging the redis container from it:

    $ docker container exec moby-counter2 ping -c 3 redis

In my case, I get a response, but it is not the one I am expecting:

    PING redis (92.242.132.15): 56 data bytes
    64 bytes from 92.242.132.15: seq=0 ttl=37 time=0.314 ms
    64 bytes from 92.242.132.15: seq=1 ttl=37 time=0.211 ms
    64 bytes from 92.242.132.15: seq=2 ttl=37 time=0.233 ms

    --- redis ping statistics ---
    3 packets transmitted, 3 packets received, 0% packet loss
       round-trip min/avg/max = 0.211/0.252/0.314 ms

Checking the resolv.conf file shows that the same nameserver is being used:

    $ docker container exec moby-counter2 cat /etc/resolv.conf

    nameserver 127.0.0.11
    options ndots:0

However, when performing a lookup, we cannot resolve redis and instead, it is falling back to my ISP's default nameservers, which are basically returning an error:

    $ docker container exec moby-counter2 nslookup redis 127.0.0.11
    Server: 127.0.0.11
    Address 1: 127.0.0.11

    Name: redis
    Address 1: 92.242.132.15 unallocated.barefruit.co.uk
Managing Containers

Let's look at launching a second Redis server in our second network; as we have already discussed, we cannot have two containers with the same name, so let's creatively name it **redis2**.

As our application is configured to connect to a container that resolves to **redis**, does this mean we will have to make changes to our application container? No, Docker has you covered.

While you cannot have two containers with the same names, as we have already discovered, our second network is running completely isolated from our first network, meaning that we can still use the DNS name of **redis**; to do this, we need to add the **--network-alias** flag:

```
$ docker container run -d --name redis2 --network moby-counter2 --network-alias redis redis:alpine
```

As you can see, we have named the container **redis2** but set the **--network-alias** to be **redis**; this means that when we perform the lookup, we see the correct IP address returned:

```
$ docker container exec moby-counter2 nslookup redis 127.0.0.11
Server: 127.0.0.11
Address 1: 127.0.0.11

Name: redis
Address 1: 172.19.0.3 redis2.moby-counter2
```

As you can see, **redis** is actually an alias for **redis2.moby-counter2**, which then resolves to **172.19.0.3**.

Now we should have two applications running side by side in their own isolated networks on your local Docker host, accessible at **http://localhost:8080/** and **http://localhost:9090/**. **Running docker network ls** will display all of the networks configured on your Docker host, including the default networks:

```
   NETWORK ID   NAME          DRIVER    SCPE
732497f548d5  bridge       bridge    local
89729949d79f  host         host      local
adf8d6f87ecb  moby-counter  bridge    local
f466b56b8be   moby-counter2 bridge    local
e7b8ac4791c   none          null      local
```

---

As you can see, the command `docker container run -d --name redis2 --network moby-counter2 --network-alias redis redis:alpine` successfully launches a second Redis server in our second network with the alias `redis2`. We can then use the `nslookup` command to verify that `redis` resolves to `redis2.moby-counter2`, which in turn resolves to `172.19.0.3`. This setup allows us to run two separate Redis services without conflicts in the same network, providing isolation and flexibility in our Docker environment.
Managing Containers

You can find out more information about the configuration of the networks by running the `inspect` command:

```
$ docker network inspect moby-counter
```

It returns the following JSON array:

```
{
  "Name": "moby-counter",
  "Id": "adfd04f86ceb0e0138b3c46c5ffac98810a572895583af0a5fce644378ce16c8",
  "Created": "2017-06-24T21:45:59.109846671Z",
  "Scope": "local",
  "Driver": "bridge",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": {},
    "Config": [
      {
        "Subnet": "172.18.0.0/16",
        "Gateway": "172.18.0.1"
      }
    ],
    "Internal": false,
    "Attachable": false,
    "Containers": {
      "1bece530450ab46450d185be2def9d7820ba3da36948cb77f5b51a931bb58738": {
        "Name": "redis",
        "EndpointID": "6f5199ec31cd852f7eff961fa9afdf94810c602af69d5a870f6209bbaed087b56",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
      },
      "8ed5183ed8d1c904b86c279ab15ac44ffe15c2a57f1fb2287170f221dc320633": {
        "Name": "moby-counter",
        "EndpointID": "fcad0cbb749b35b5ead29d424919f6549c11a3a0364116d2d6e1d02b217411",
        "MacAddress": "02:42:ac:12:00:03",
        "IPv4Address": "172.18.0.3/16",
        "IPv6Address": ""
      }
    }
  }
}
```
As you can see, it contains information on the network addressing being used in the IPAM section, and also details on each of the two containers running in the network.

**IP address management (IPAM)** is a means of planning, tracking, and managing IP addresses within the network. IPAM has both DNS and DHCP services so that each service is notified of changes in the other; for example, DHCP assigns an address to container2. The DNS service is updated to return the IP address assigned by DHCP whenever a lookup is made against container2.

Before we progress to the next section, we should remove one of the applications and associated networks. To do this, run the following commands:

```
$ docker container stop moby-counter2 redis2
$ docker container prune
$ docker network prune
```

This will remove the containers and network:
As mentioned at the start of this section, this is only the default network driver, meaning that we are restricted to our networks only being available on a single Docker host. In later chapters, we will look at how we can expand our Docker network across multiple hosts and even providers.

**Docker volumes**

If you have been following along with the network example from the previous section, you should have two containers running:

When you go to the application in a browser (at http://localhost:8080/), you will probably see that there already are Docker logos on screen. Let's stop and remove the Redis container and see what happens. To do this, run the following:

```bash
$ docker container stop redis
$ docker container rm redis
```

If you have your browser open, you may notice that the Docker icons have faded into the background and there is an animated loader in the center of the screen; this is basically to show that the application is waiting for the connection to the Redis container to be reestablished:
Relaunch the Redis container using the following command:

```
$ docker container run -d --name redis --network moby-counter redis:alpine
```
Managing Containers

This restores the connectivity; however, when you start to interact with the application, your previous icons disappear and you are left with a clean slate. Quickly add some more logos to the screen, this time placed in a different pattern like I have done here:

Once you have a pattern, let’s remove the Redis container again by running:

```
$ docker container stop redis
$ docker container rm redis
```

As we discussed earlier in the chapter, losing the data in the container is to be expected. However, as we used the official Redis image from https://store.docker.com/images/redis/, we haven't in fact lost any of our data.

The Dockerfile for the Redis official image we used looks like the following:

```
FROM alpine:3.5

RUN addgroup -S redis && adduser -S -G redis redis
RUN apk add --no-cache 'su-exec>=0.2'

ENV REDIS_VERSION 3.0.7
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-3.0.7.tar.gz
ENV REDIS_DOWNLOAD_SHA e56b4b7e033ae8df311f9191cf6f6df3ae9ff4d1c
```
If you notice, toward the end of the file, there are the `VOLUME` and `WORKDIR` directives declared; this means that when our container was launched, Docker actually created a volume and then ran `redis-server` from within the volume.

We can see this by running the following command:

```
$ docker volume ls
```

This should show at least two volumes:
As you can see, the volume name is not very friendly at all, in fact, it is the unique ID of the volume. So how can we use the volume when we launch our Redis container? We know from the Dockerfile that the volume was mounted at /data within the container, so all we have to do is tell Docker which volume to use and where it should be mounted at runtime.

To do this, run the following command, making sure you replace the volume ID with that of your own:

```
$ docker container run -d --name redis -v
719d0cc415dbc76fed5e9b8893e2cf547f0ac6c91233451604fdba31f0dd2d2a:/data
--network moby-counter redis:alpine
```

If your application page looks like it is still trying to reconnect to the Redis container once you have launched your Redis container, then you may need to refresh your browser; failing that, restarting the application container by running `docker container restart moby-counter` and then refreshing your browser again should work.

You should be able to see the icons in their original positions. We can view the contents of the /data folder on the Redis container by running:

```
$ docker container exec redis ls -lhat /data
```

This will return something that looks like the following:

```
total 12
  drwxr-xr-x 1 root root 4.0K Jun 25 14:55 ..
  drwxr-xr-x 2 redis redis 4.0K Jun 25 14:00 .
  -rw-r--r-- 1 redis redis 421 Jun 25 14:00 dump.rdb
```

You can also remove your running container and relaunch it, but this time using the ID of the second volume.

Finally, you can override the volume with your own. To create a volume, we need to use the `volume` command:

```
$ docker volume create redis_data
```

Once created, we will be able to use the `redis_data` volume to store our Redis on by running this command:

```
$ docker container run -d --name redis -v redis_data:/data --network moby-counter redis:alpine
```
Managing Containers

We can then use it as needed:

Like the `network` command, we can view more information on the volume using the `inspect` command:

```
$ docker volume inspect redis_data
```

Look at the following output:

```
[
  {
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/redis_data/_data",
    "Name": "redis_data",
    "Options": {},
    "Scope": "local"
  }
]
```

You can see that there is not much to a volume when using the local driver; one interesting thing to note is that the path to where the data is stored on the Docker host machine is `/var/lib/docker/volumes/redis_data/_data`. If you are using Docker for Mac or Docker for Windows, then this path will be your Docker host VM and not your local machine, meaning that you do not have direct access to the data inside the volume.
Managing Containers

Don't worry though; we will be looking at Docker volumes and how you can interact with data in later chapters. For now, we should tidy up. First of all, remove the two containers and network:

```
$ docker container stop redis moby-counter
$ docker container prune
$ docker network prune
```

Then we can remove the volumes by running the following:

```
$ docker volume prune
```

You should see something similar to the following Terminal output:

```
1. russ (bash)

$ docker container stop redis moby-counter
- redis
- moby-counter

WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
- 177937e2f058bb424d7ba461b4257fd89e41f61f465c4b0b09881b200e5a0f65
- 8ed5183adedd1c904b86c279ab15ac44ffe15c2a57f1fb2287178f221dc320633

Total reclaimed space: 0 B

$ docker network prune

WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Networks:
- moby-counter

$ docker volume prune

WARNING! This will remove all volumes not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Volumes:
- 719d80cc415dabc76fed5e9b8893e2cf547f0ac6c91233451604fdba3170dd2d2a
- d6107f7e25b55011f1f144e9030ee44148b463c20925458bf68085e78fee54b
- redis_data

Total reclaimed space: 743 B
```

As you can see, we are now back to having a clean slate.
Summary

In this chapter, we looked at how you can use the Docker command-line client to manage both individual containers as well as launch multi-container applications in their own isolated Docker networks. We also discussed how we can persist data on the filesystem using Docker volumes.

So far in this and the previous sections, we have covered in detail the majority of the available commands used in the following sections:

\$ docker container [command]
\$ docker network [command]
\$ docker volume [command]
\$ docker image [command]

We have covered the four main areas of using Docker locally; we can now start to look at how we can interact with remote Docker hosts.