Linking Containers Together

In the Using Docker section (/userguide/usingdocker), you saw how you can connect to a service running inside a Docker container via a network port. But a port connection is only one way you can interact with services and applications running inside Docker containers. In this section, we'll briefly revisit connecting via a network port and then we'll introduce you to another method of access: container linking.

Network port mapping refresher

In the Using Docker section (/userguide/usingdocker), you created a container that ran a Python Flask application:

```
$ sudo docker run -d -P training/webapp python app.py
```

> ***Note:*** *Containers have an internal network and an IP address (as we saw when we used the* `docker inspect` *command to show the container's IP address in the Using Docker (/userguide/usingdocker/) section). Docker can have a variety of network configurations. You can see more information on Docker networking here (/articles/networking/).*

When that container was created, the `-P` flag was used to automatically map any network ports inside it to a random high port from the range 49153 to 65535 on our Docker host. Next, when `docker ps` was run, you saw that port 5000 in the container was bound to port 49155 on the host.

```
$ sudo docker ps nostalgic_morse
CONTAINER ID  IMAGE                  COMMAND       CREATED        STATUS        PORTS                    NAMES
bc533791f3f5  training/webapp:latest  python app.py 5 seconds ago  Up 2 seconds  0.0.0.0:49155->5000/tcp  nostalgic_morse
```

You also saw how you can bind a container's ports to a specific port using the `-p` flag:

```
$ sudo docker run -d -p 5000:5000 training/webapp python app.py
```

And you saw why this isn't such a great idea because it constrains you to only one container on that specific port.

There are also a few other ways you can configure the `-p` flag. By default the `-p` flag will bind the specified port to all interfaces on the host machine. But you can also specify a binding to a specific interface, for example only to the `localhost`.

```
$ sudo docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py
```

This would bind port 5000 inside the container to port 5000 on the `localhost` or `127.0.0.1` interface on the host machine.

Or, to bind port 5000 of the container to a dynamic port but only on the `localhost`, you could use:

```
$ sudo docker run -d -p 127.0.0.1::5000 training/webapp python app.py
```

You can also bind UDP ports by adding a trailing `/udp`. For example:

```
$ sudo docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
```

You also learned about the useful `docker port` shortcut which showed us the current port bindings. This is also useful for showing you specific port configurations. For example, if you've bound the container port to the `localhost` on the host machine, then the `docker port` output will reflect that.

```
$ sudo docker port nostalgic_morse 5000
127.0.0.1:49155
```

> ***Note:*** *The* `-p` *flag can be used multiple times to configure multiple ports.*

Docker Container Linking

Network port mappings are not the only way Docker containers can connect to one another. Docker also has a linking system that allows you to link multiple containers together and send connection information from one to another. When containers are linked, information about a source container can be sent to a recipient container. This allows the recipient to see selected data describing aspects of the source container.

Container naming

To establish links, Docker relies on the names of your containers. You've already seen that each container you create has an automatically created name; indeed you've become familiar with our old friend `nostalgic_morse` during this guide. You can also name containers yourself. This naming provides two useful functions:

1. It can be useful to name containers that do specific functions in a way that makes it easier for you to remember them, for example naming a container containing a web application `web`.
2. It provides Docker with a reference point that allows it to refer to other containers, for example, you can specify to link the container `web` to container `db`.

You can name your container by using the `--name` flag, for example:

```
$ sudo docker run -d -P --name web training/webapp python app.py
```

This launches a new container and uses the `--name` flag to name the container `web`. You can see the container's name using the `docker ps` command.

```
$ sudo docker ps -l
CONTAINER ID  IMAGE                  COMMAND        CREATED       STATUS       PORTS                    NAMES
aed84ee21bde  training/webapp:latest python app.py  12 hours ago  Up 2 seconds 0.0.0.0:49154->5000/tcp  web
```

You can also use `docker inspect` to return the container's name.

```
$ sudo docker inspect -f "{{ .Name }}" aed84ee21bde
/web
```

> ***Note:*** *Container names have to be unique. That means you can only call one container `web`. If you want to re-use a container name you must delete the old container (with `docker rm`) before you can create a new container with the same name. As an alternative you can use the `--rm` flag with the `docker run` command. This will delete the container immediately after it is stopped.*

Container Linking

Links allow containers to discover each other and securely transfer information about one container to another container. When you set up a link, you create a conduit between a source container and a recipient container. The recipient can then access select data about the source. To create a link, you use the `--link` flag. First, create a new container, this time one containing a database.

```
$ sudo docker run -d --name db training/postgres
```

This creates a new container called `db` from the `training/postgres` image, which contains a PostgreSQL database.

Now, you need to delete the `web` container you created previously so you can replace it with a linked one:

```
$ sudo docker rm -f web
```

Now, create a new `web` container and link it with your `db` container.

```
$ sudo docker run -d -P --name web --link db:db training/webapp python app.py
```

This will link the new `web` container with the `db` container you created earlier. The `--link` flag takes the form:

```
--link name:alias
```

Where `name` is the name of the container we're linking to and `alias` is an alias for the link name. You'll see how that alias gets used shortly.

Next, inspect your linked containers with `docker inspect`:

```
$ sudo docker inspect -f "{{ .HostConfig.Links }}" web
[/db:/web/db]
```

You can see that the `web` container is now linked to the `db` container `web/db`. Which allows it to access information about the `db` container.

So what does linking the containers actually do? You've learned that a link allows a source container to provide information about itself to a recipient container. In our example, the recipient, `web`, can access information about the source `db`. To do this, Docker creates a secure tunnel between the containers that doesn't need to expose any ports externally on the container; you'll note when we started the `db` container we did not use either the `-P` or `-p` flags. That's a big benefit of linking: we don't need to expose the source container, here the PostgreSQL database, to the network.

Docker exposes connectivity information for the source container to the recipient container in two ways:

- Environment variables,
- Updating the `/etc/hosts` file.

Environment Variables

When two containers are linked, Docker will set some environment variables in the target container to enable programmatic discovery of information related to the source container.

First, Docker will set an `<alias>_NAME` environment variable specifying the alias of each target container that was given in a `--link` parameter. So, for example, if a new container called `web` is being linked to a database container called `db` via `--link db:webdb` then in the `web` container would be `WEBDB_NAME=/web/webdb`.

Docker will then also define a set of environment variables for each port that is exposed by the source container. The pattern followed is:

- `<name>_PORT_<port>_<protocol>` will contain a URL reference to the port. Where `<name>` is the alias name specified in the `--link` parameter (e.g. `webdb`), `<port>` is the port number being exposed, and `<protocol>` is either `TCP` or `UDP`. The format of the URL will be: `<protocol>://<container_ip_address>:<port>` (e.g. `tcp://172.17.0.82:8080`). This URL will then be split into the following 3 environment variables for convenience:
- `<name>_PORT_<port>_<protocol>_ADDR` will contain just the IP address from the URL (e.g. `WEBDB_PORT_8080_TCP_ADDR=172.17.0.82`).
- `<name>_PORT_<port>_<protocol>_PORT` will contain just the port number from the URL (e.g. `WEBDB_PORT_8080_TCP_PORT=8080`).
- `<name>_PORT_<port>_<protocol>_PROTO` will contain just the protocol from the URL (e.g. `WEBDB_PORT_8080_TCP_PROTO=tcp`).

If there are multiple ports exposed then the above set of environment variables will be defined for each one.

Finally, there will be an environment variable called `<alias>_PORT` that will contain the URL of the first exposed port of the source container. For example, `WEBDB_PORT=tcp://172.17.0.82:8080`. In this case, 'first' is defined as the lowest numbered port that is exposed. If that port is used for both tcp and udp, then the tcp one will be specified.

Returning back to our database example, you can run the `env` command to list the specified container's environment variables.

```
$ sudo docker run --rm --name web2 --link db:db training/webapp env
. . .
DB_NAME=/web2/db
DB_PORT=tcp://172.17.0.5:5432
DB_PORT_5432_TCP=tcp://172.17.0.5:5432
DB_PORT_5432_TCP_PROTO=tcp
DB_PORT_5432_TCP_PORT=5432
DB_PORT_5432_TCP_ADDR=172.17.0.5
. . .
```

> ***Note**: These Environment variables are only set for the first process in the container. Similarly, some daemons (such as `sshd`) will scrub them when spawning shells for connection.*
> ***Note**: Unlike host entries in the `/ets/hosts` file, IP addresses stored in the environment variables are not automatically updated if the source container is restarted. We recommend using the host entries in `/etc/hosts` to resolve the IP address of linked containers.*

You can see that Docker has created a series of environment variables with useful information about the source `db` container. Each variable is prefixed with `DB_`, which is populated from the `alias` you specified above. If the `alias` were `db1`, the variables would be prefixed with `DB1_`. You can use these environment variables to configure your applications to connect to the database on the `db` container. The connection will be secure and private; only the linked `web` container will be able to talk to the `db` container.

Updating the `/etc/hosts` file

In addition to the environment variables, Docker adds a host entry for the source container to the `/etc/hosts` file. Here's an entry for the `web` container:

```
$ sudo docker run -t -i --rm --link db:db training/webapp /bin/bash
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7  aed84ee21bde
. . .
172.17.0.5  db
```

You can see two relevant host entries. The first is an entry for the `web` container that uses the Container ID as a host name. The second entry uses the link alias to reference the IP address of the `db` container. You can ping that host now via this host name.

```
root@aed84ee21bde:/opt/webapp# apt-get install -yqq inetutils-ping
root@aed84ee21bde:/opt/webapp# ping db
PING db (172.17.0.5): 48 data bytes
56 bytes from 172.17.0.5: icmp_seq=0 ttl=64 time=0.267 ms
56 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.250 ms
56 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.256 ms
```

> ***Note:*** *In the example, you'll note you had to install* `ping` *because it was not included in the container initially.*

Here, you used the `ping` command to ping the `db` container using its host entry, which resolves to `172.17.0.5`. You can use this host entry to configure an application to make use of your `db` container.

> ***Note:*** *You can link multiple recipient containers to a single source. For example, you could have multiple (differently named) web containers attached to your* `db` *container.*

If you restart the source container, the linked containers `/etc/hosts` files will be automatically updated with the source container's new IP address, allowing linked communication to continue.

```
$ sudo docker restart db
root@aed84ee21bde:/opt/webapp# cat /etc/hosts
172.17.0.7  aed84ee21bde
. . .
172.17.0.9  db
```

Next step

Now that you know how to link Docker containers together, the next step is learning how to manage data, volumes and mounts inside your containers.

Go to Managing Data in Containers (/userguide/dockervolumes).