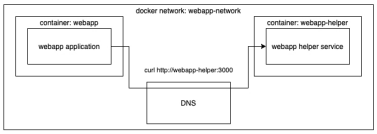
Docker networking

Connecting two services such as a server and its database in docker can be achieved with a Docker network. In addition to starting services listed in docker-compose.yml Docker Compose automatically creates and joins both containers into a network with a DNS. Each service is named after the name given in the docker-compose.yml file. As such, containers can reference each other simply with their service names, which is different from the container name.



Here are two services in a single network: webapp and webapp-helper. The webapp-helper has a server, listening for requests in port 3000, that webapp wants to access. Because they were defined in the same docker-compose.yml file the access is trivial. Docker Compose has already taken care of creating a network and webapp can simply send a request to webapp-helper:3000, the internal DNS will translate that to the correct access and ports do not have to be published outside of the network.

SECURITY REMINDER: PLAN YOUR INFRASTRUCTURE AND KEEP TO YOUR PLAN

In the next exercise, and in some later exercises, there is an illustration of the infrastructure. Have a look at it and use it to write the configuration.

For example, in Exercise 2.4 we don't want to open ports to Redis to the outside world. Do not add a `ports` configuration under Redis! The backend will be able to access the application within the Docker network.

Exercise 2.4

EXERCISE 2.4

In this exercise you should expand the configuration done in Exercise 2.3 and set up the example backend to use the key-value database Redis.

Redis is quite often used as a cache to store data so that future requests for data can be served faster.

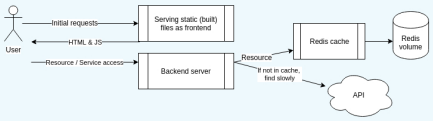
The backend uses a slow API to fetch some information. You can test the slow API by requesting `/ping?redis=true` with curl. The frontend app has a button to test this.

So you should improve the performance of the app and configure a Redis container to cache information for the backend. The documentation of the Redis image might contain some useful info.

The backend README should have all the information that is needed for configuring the backend.

When you've correctly configured the button will turn green.

Submit the docker-compose.yml



The restart: unless-stopped configuration can help if the Redis takes a while to get ready.

Manual network definition

It is also possible to define the network manually in a Docker Compose file. A major benefit of a manual network definition is that it makes it easy to set up a configuration where containers defined in two different Docker Compose files share a network, and can easily interact with each other.

Let us now have a look how a network is defined in docker-compose.yml:

```
version: "3.8"

services:
  db:
    image: postgres:13.2-alpine
    networks:
      - database-network # Name in this Docker Compose file

networks:
  database-network: # Name in this Docker Compose file
    name: database-network # Name that will be the actual name of the network
```

This defines a network called `database-network` which is created with `docker compose up` and removed with `docker compose down`.

As can be seen, services are configured to use a network by adding `networks` into the definition of the service.

Establishing a connection to an external network (that is, a network defined in another docker-compose.yml, or by some other means) is done as follows:

```
version: "3.8"

services:
  db:
    image: backend-image
    networks:
      - database-network

networks:
  database-network:
    external:
      name: database-network # Must match the actual name of the network
```

By default all services are added to a network called `default`. The default network can be configured and this makes it possible to connect to an external network by default as well:

```
version: "3.8"

services:
  db:
    image: backend-image

networks:
  default:
    external:
      name: database-network # Must match the actual name of the network
```

Scaling

Compose can also scale the service to run multiple instances:

```
$ docker compose up --scale whoami=3

  WARNING: The "whoami" service specifies a port on the host. If multiple containers for this service a

  Starting whoami_whoami_1 ... done
  Creating whoami_whoami_2 ... error
```

```
Creating whoami_whoami_2 ... error
Creating whoami_whoami_3 ... error
```

The command fails due to a port clash, as each instance will attempt to bind to the same host port (8000).

We can get around this by only specifying the container port. As mentioned in part 1, when leaving the host port unspecified, Docker will automatically choose a free port.

Update the ports definition in `docker-compose.yml`:

```
ports:
  - 8000
```

Then run the command again:

```
$ docker compose up --scale whoami=3
  Starting whoami_whoami_1 ... done
  Creating whoami_whoami_2 ... done
  Creating whoami_whoami_3 ... done
```

All three instances are now running and listening on random host ports. We can use `docker compose port` to find out which ports the instances are bound to.

```
$ docker compose port --index 1 whoami 8000
  0.0.0.0:32770

$ docker compose port --index 2 whoami 8000
  0.0.0.0:32769

$ docker compose port --index 3 whoami 8000
  0.0.0.0:32768
```

We can now curl from these ports:

```
$ curl 0.0.0.0:32769
  I'm 536e11304357

$ curl 0.0.0.0:32768
  I'm 1ae20cd990f7
```

In a server environment you'd often have a load balancer in front of the service. For containerized local environment (or a single server) one good solution is to use https://github.com/jwilder/nginx-proxy.

Let's add the nginx-proxy to our compose file and remove the port bindings from the whoami service. We'll mount our docker.sock (the socket that is used to communicate with the Docker Daemon) inside of the container in `:ro` read-only mode:

```
version: "3.8"

services:
  whoami:
    image: jwilder/whoami
  proxy:
    image: jwilder/nginx-proxy
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock:ro
    ports:
      - 80:80
```

Let test the configuration:

```
$ docker compose up -d --scale whoami=3
$ curl localhost:80
  <html>
  <head><title>503 Service Temporarily Unavailable</title></head>
  <body bgcolor="white">
  <center><h1>503 Service Temporarily Unavailable</h1></center>
  <hr><center>nginx/1.13.8</center>
  </body>
  </html>
```

It's "working", but the Nginx just doesn't know which service we want. The `nginx-proxy` works with two environment variables: `VIRTUAL_HOST` and `VIRTUAL_PORT`. `VIRTUAL_PORT` is not needed if the service has `EXPOSE` in it's Docker image. We can see that `jwilder/whoami` sets it: https://github.com/jwilder/whoami/blob/master/Dockerfile#L9

- Note: Mac users with the M1 processor you may see the following error message: `runtime: failed to create new OS thread`. In this case you can use the Docker Image `ninanung/nginx-proxy` instead which offers a temporary fix until `jwilder/nginx-proxy` is updated to support M1 Macs.

The domain `colasloth.com` is configured so that all subdomains point to `127.0.0.1`. More information about how this works can be found at colasloth.github.io, but in brief it's a simple DNS "hack". Several other domains serving the same purpose exist, such as `localtest.me`, `lvh.me`, and `vcap.me`, to name a few. In any case, let's use `colasloth.com` here:

```
version: "3.8"

services:
  whoami:
    image: jwilder/whoami
    environment:
      - VIRTUAL_HOST=whoami.colasloth.com
  proxy:
    image: jwilder/nginx-proxy
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock:ro
    ports:
      - 80:80
```

Now the proxy works:

```
$ docker compose up -d --scale whoami=3
$ curl whoami.colasloth.com
  I'm f6f85f4848a8
$ curl whoami.colasloth.com
  I'm 740dc0de1954
```

Let's add couple of more containers behind the same proxy. We can use the official `nginx` image to serve a simple static web page. We don't have to even build the container images, we can just mount the content to the image. Let's prepare some content for two services called "hello" and "world".

```
$ echo "hello" > hello.html
$ echo "world" > world.html
```

Then add these services to the `docker-compose.yml` file where you mount just the content as `index.html` in the default nginx path:

```
hello:
  image: nginx:1.19-alpine
  volumes:
    - ./hello.html:/usr/share/nginx/html/index.html:ro
  environment:
    - VIRTUAL_HOST=hello.colasloth.com
world:
  image: nginx:1.19-alpine
  volumes:
    - ./world.html:/usr/share/nginx/html/index.html:ro
  environment:
    - VIRTUAL_HOST=world.colasloth.com
```

Now let's test:

```
$ docker compose up -d --scale whoami=3
$ curl hello.colasloth.com
  hello

$ curl world.colasloth.com
```

```
  world

$ curl whoami.colasloth.com
  I'm f6f85f4848a8

$ curl whoami.colasloth.com
  I'm 740dc0de1954
```

Now we have a basic single-machine hosting setup up and running.

Test updating the `hello.html` without restarting the container, does it work?

Exercises 2.5

> **EXERCISE 2.5**
>
> The project https://github.com/docker-hy/material-applications/tree/main/scaling-exercise is a barely working application. Go ahead and clone it for yourself. The project already includes docker-compose.yml so you can start it by running `docker compose up`.
>
> The application should be accessible through http://localhost:3000. However it doesn't work well enough and we've added a load balancer for scaling. Your task is to scale the `compute` containers so that the button in the application turns green.
>
> This exercise was created with Sasu Mäkinen
>
> Please return the used commands for this exercise.

Edit this page

Previous
« Migrating to Docker Compose

Next
Volumes in action »