Assign Static IP to Docker Container and Docker-Compose

---

1. Overview

When we run a Docker container, it connects with a virtual network using an IP address. Thus, we usually expect services to get a configuration dynamically. However, we might want to use a static IP instead of an automatic IP allocation.

Still, **if we think we need a static, private IP address, we should consider that need in the first place**. Most of the time, we want a static IP to talk to one container from another or the host. Docker's built-in networking can already handle this. It's worth considering the possibility of custom networking using Docker Swarm. However, we might want to manually specify a private IP address, for example, for accessing containers directly from the host.

In this tutorial, we'll see the difference between the built-in configuration and assigning a manual static IP to a container. Finally, we'll add some Docker Compose examples with tests.

2. Docker DHCP and DNS

To begin with, let's explore the built-in Docker IP assignment to containers using DHCP and DNS to resolve host names.

Docker first assigns an IP to each container, acting as a DHCP server.

Containers then process DNS requests through a server inside *dockerd*, which recognizes the names of other containers on the same internal network. This way, containers can communicate without knowing their internal IP addresses. Although each time the internal IP addresses might differ when the application starts, containers can still easily connect with a human-readable name thanks to the internal DNS server inside *dockerd*.

Further, ***dockerd* sends name lookups to CoreDNS (from the CNCF)**. Finally, requests move to the host depending on the domain name.

There's a side case for the *docker.internal* domain. It includes the DNS name *host.docker.internal* that resolves to a valid IP address for the current host. It enables containers to contact those host services without worrying about hardcoding IP addresses. Although not recommended, *docker.internal* can be handy for development purposes.

3. Docker Network Example

As an example, we can run a container for a MySQL service. Let's check out the Docker Compose YAML definition:

```
$ cat docker-compose.yml
services:
  db:
    image: mysql:latest
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_ROOT_HOST=localhost
    ports:
      - 3306:3306
    volumes:
      - db:/var/lib/mysql
    networks:
      - network

volumes:
  db:
    driver: local

networks:
  network:
    driver: bridge
```

As usual, we run our container:

```
$ docker compose up -d
...
✓ Container compo-db-1 Started
```

Let's inspect the network from a container perspective with the *format* syntax using *jq* to get a JSON output:

```
$ docker inspect --format='{{json .NetworkSettings.Networks}}' compo-db-1 |
jq .
```

Docker Compose assigns the network name based on the current directory. We can see a similar output if, for example, we are in the *project* directory:

```
{
  "project_network": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": [
      "project-db-1",
      "db",
      "2d3f4c69a213"
    ],
    "NetworkID":
"39ffbd8155d11ba03d0b548307f549f06790fe045e121a6d862b070d4fb67fa7",
    "EndpointID":
"0eba235239b06f7e0cb5065b7f2ebd83e7d227f8cfad4df8de73260472737500",
```

```
    "Gateway": "172.19.0.1",
    "IPAddress": "172.19.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:ac:13:00:02",
    "DriverOpts": null
  }
}
```

The container gets a private *172.19.0.2* IP address from the subnet created by the network.

Most importantly, we can see info about *IPAMConfig*, i.e., IP address management. This is relevant when we statically assign the IP address.

Now, we can inspect the network itself:

```
$ docker inspect project_network
```

This time, we have a better insight into the network:

```
[
    {
        "Name": "project_network",
        "Id":
"39ffbd8155d11ba03d0b548307f549f06790fe045e121a6d862b070d4fb67fa7",
        "Created": "2022-09-09T16:19:26.27396468+02:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {

"2d3f4c69a2139dea9089a6d42907fdc085282c5df176b39bf7c20f5d0780179d": {
                "Name": "project-db-1",
                "EndpointID":
"7447fe2550afb3f980f36449673724e9ed6dd16f41a085cc20ada3074a0d8e54",
```

```
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
            "com.docker.compose.network": "network",
            "com.docker.compose.project": "project",
            "com.docker.compose.version": "2.10.2"
        }
    }
]
```

It's worth noting the Docker Compose network has been available since version 2.

Armed with a bit more knowledge about automatic IP assignment, we can now create a specific network. After that, we assign the IP address of preference to the containers.

4.1. Assign a Static IP via Docker Directly

When using Docker CLI, **we first create a network**:

```
$ docker network create --subnet=10.5.0.0/16 custom_net
```

At this point, we have the *custom_net* network with a subnet of *10.5.0.0/16*.

Then, we can run a container with a static IP:

```
$ docker run --detach --net custom_net --ip 10.5.0.5 -p 3306:3306 --mount
source=db,target=/var/lib/mysql -e MYSQL_ROOT_PASSWORD=password mysql:latest
```

Notably, **we supply the *custom_net* network name to the relevant *–net* option along with the static IP as *10.5.0.5* after *–ip***.

4.2. Assign a Static IP via Docker Compose

Of course, we can also do the above through Docker Compose:

```
$ cat docker-compose.yml
services:
  db:
    container_name: mysql_db
    image: mysql:latest
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_ROOT_HOST=10.5.0.1
    ports:
      - 3306:3306
    volumes:
      - db:/var/lib/mysql
```

```
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      custom_net:
        ipv4_address: 10.5.0.5

volumes:
  db:
    driver: local

networks:
  custom_net:
    driver: bridge
    ipam:
      config:
        - subnet: 10.5.0.0/16
          gateway: 10.5.0.1
```

Thus, **we define the network subnet under the *ipam* keyword and assign an IPv4 address to the service via *ipv4_address* within the service definition**. For coherency, we use the same *10.5.0.5* IP address. Commonly, 172.* and *10.** IP addresses are ones chosen for Docker private networks. Of course, we can also use an IPv6 address, which has a 128-bit address length and gradually replaces IPv4 due to more efficiency.

As recommended, **we assign the *10.5.0.1* host and gateway address to *MYSQL_ROOT_HOST* to allow direct connections**.

4.3. Verification

Finally, we can run an SQL script to create a user, a database, and a table:

```
CREATE DATABASE IF NOT EXISTS test;
CREATE USER 'db_user'@'10.5.0.1' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON *.* TO 'db_user'@'10.5.0.1' WITH GRANT OPTION;
FLUSH PRIVILEGES;

use test;

CREATE TABLE IF NOT EXISTS TEST_TABLE (id int, name varchar(255));

INSERT INTO TEST_TABLE VALUES (1, 'TEST_1');
INSERT INTO TEST_TABLE VALUES (2, 'TEST_2');
INSERT INTO TEST_TABLE VALUES (3, 'TEST_3');
```

We want to give the user access to the database only at that specific address.

After the container starts, we can have a look at its definition with *docker ps*:

```
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS
PORTS                                                  NAMES
97812e199512   mysql:latest   "docker-entrypoint.s…"   7 minutes ago   Up 7
minutes   0.0.0.0:3306->3306/tcp, :::3306->3306/tcp, 33060/tcp   mysql_db
```

We can now connect to the database by entering the password. We use the container name or ID as these resolve as an alias for DNS:

```
$ mysql --host=mysql_db -u db_user -p
```

Now, using the *status* command, we can verify the MySQL host resolves as the container ID:

```
Connection id:          10
Current database:       test
Current user:           db_user@10.5.0.1
SSL:                    Not in use
Current pager:          stdout
Using outfile:          ''
Using delimiter:        ;
Server:                 MySQL
Server version:         8.0.30 MySQL Community Server - GPL
Protocol version:       10
Connection:             97812e199512 via TCP/IP
Server characterset:    utf8mb4
Db     characterset:    utf8mb4
Client characterset:    utf8mb3
Conn.  characterset:    utf8mb3
TCP port:               3306
```

So, our setup looks good.

4.4. Container Inspection

Let's inspect the container:

```
$ docker inspect mysql_db
```

In the case of a static IP, we can see that the *IPAM* configuration now has an IPv4 address:

```
{
  "project_custom_net": {
    "IPAMConfig": {
      "IPv4Address": "10.5.0.5"
    },
    "Links": null,
    "Aliases": [
      "mysql_db",
      "db",
      "122c0c6bfcf9"
    ],
    "NetworkID":
"7ac7a1d9e33dffc65bc867aee4db04b9b8fecaeb3bbb91c74c2f72e4611c6955",
    "EndpointID":
"84145191a0327b777b6a31bacb2a0260d9a31e8c22cbfca1923775b3649b1d7e",
    "Gateway": "10.5.0.1",
    "IPAddress": "10.5.0.5",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
```

```
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:0a:05:00:05",
    "DriverOpts": null
  }
}
```

From a container perspective, this *IPv4Address* field is the main difference. However, we can also see that **the network name we specified in the Docker Compose file has been prefixed with the directory name**: *project_custom_net*.