Connecting the DB Container

Connecting the application and database

We'll cover the following

- App and DB connectivity
- Flask DB connector
- docker-entrypoint-initdb.d explanation

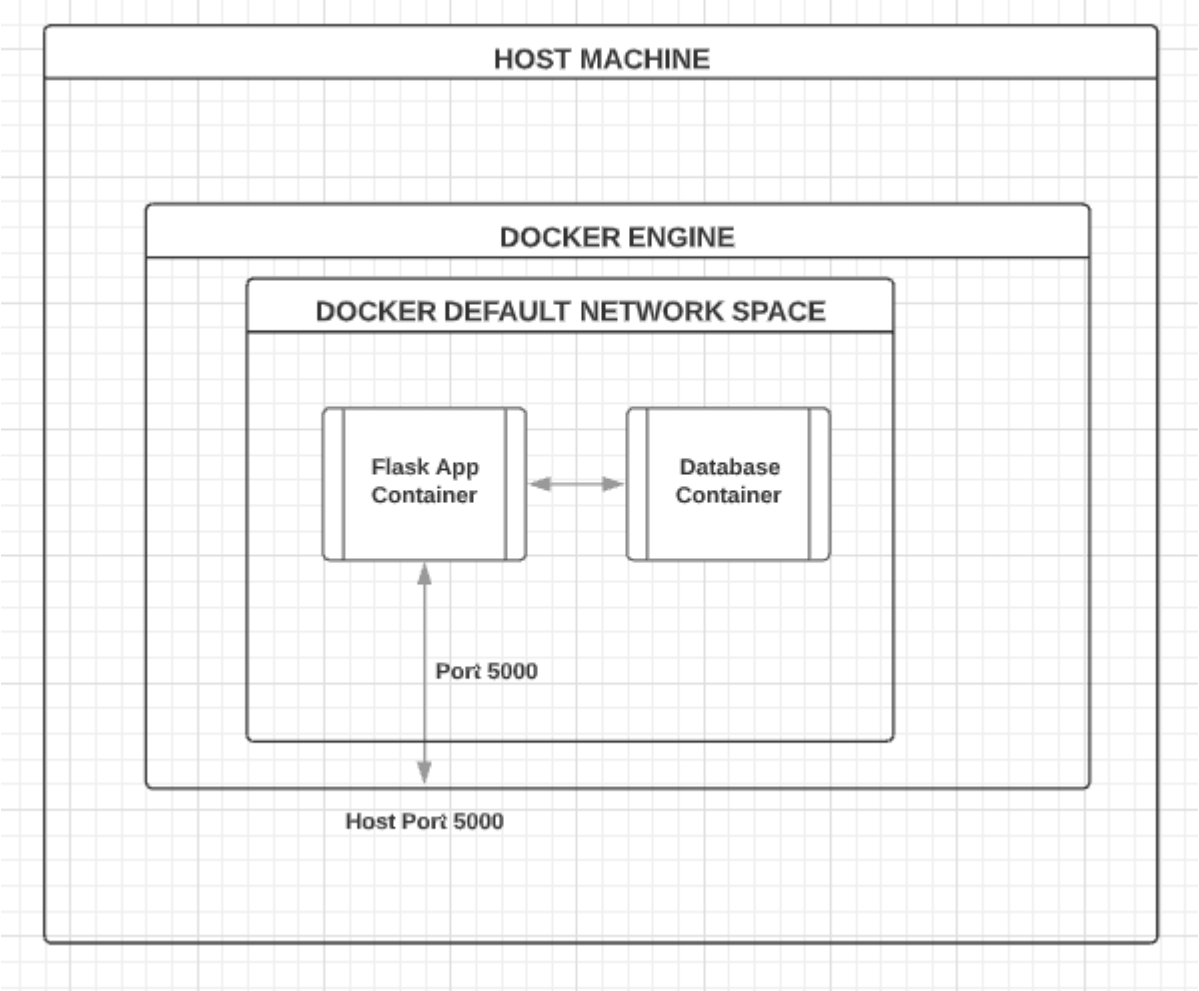
In this lesson, we will complete the remaining steps.

> For this lesson, `git checkout 6589c4f6586ca7efb25d8819fc886181dec771e9` to get to the specific code.

Till now, we have gotten to get the MySQL shell of the container and checked for our database. The next step is to establish a connection between the app and the database container.

App and DB connectivity

As per the diagram below, by default, both containers should be in the same bridge network. So, let's check for that.



Type `docker network ls` & `docker inspect <NETWORK ID>` and look for the containers section.

```
$ docker inspect 6fd2dbdb22d2
[
    {
        "Name": "bridge",
        "Id": "6fd2dbdb22d2e2fa3595da548a8260ccd700943d647ab195340cceb10b32d88
6",
        "Created": "2020-04-08T13:28:26.015892567Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
```

```
        },
        "Internal": false,

        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "029acceda0674c23a09c7f37b7c3af733cc295f429573e8ef266882d6fc3912a"
: {
                "Name": "mysql",
                "EndpointID": "f8184cbe809537ec80902323655c6f7ee9fd31940c60125
2f34fdede1d7201fd",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },
            "57c5fdd375b5e9e5f923fbc289623c3a12995559af501d9cab8be11e788893dc"
: {
                "Name": "interesting_hypatia",
                "EndpointID": "af7da2a39e84006db2ecace93602acddaaa16b2e9603a6e
df8c4d1a007c80e17",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```


Closely look at the containers part below:


```
"Containers": {
            "029acceda0674c23a09c7f37b7c3af733cc295f429573e8ef266882d6fc3912a"
: {
```

```
                "Name": "mysql",
                "EndpointID": "f8184cbe809537ec80902323655c6f7ee9fd31940c601252f34fdede1d7201fd",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },
            "57c5fdd375b5e9e5f923fbc289623c3a12995559af501d9cab8be11e788893dc": {
                "Name": "interesting_hypatia",
                "EndpointID": "af7da2a39e84006db2ecace93602acddaaa16b2e9603a6edf8c4d1a007c80e17",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
```


Both containers are in the same network. That means there should be an entry of our database container in the /etc/hosts file of the application container. But surprisingly, there won't be any entry and the DB container will be still unreachable despite being in the same network.

```
$ docker exec -it 1ac336df8cf6 bash
root@1ac336df8cf6:/code# cat /etc/hosts
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
172.17.0.2      1ac336df8cf6
```

The reason is bridge network needs containers to be 'linked' to make it reachable. We discussed this in the 'Introduction to Docker networks' lesson.

So, let's link the DB container to our application container. First, stop the application container and rerun with:

`docker run --link "mysql:backenddb" -p 5000:5000 flask_app:1.0` and then, check.

```
$ docker exec -it f9cf42e23825 bash
root@f9cf42e23825:/code# cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
```

```
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3      backenddb 029acceda067 mysql
172.17.0.2      f9cf42e23825
root@f9cf42e23825:/code# ping mysql
PING backenddb (172.17.0.3) 56(84) bytes of data.
64 bytes from backenddb (172.17.0.3): icmp_seq=1 ttl=64 time=2.01 ms
64 bytes from backenddb (172.17.0.3): icmp_seq=2 ttl=64 time=0.358 ms
64 bytes from backenddb (172.17.0.3): icmp_seq=3 ttl=64 time=0.212 ms
```

Flask DB connector

Let's do some modification in the application code and remove the temporary dictionary database.

1. `flask_mysqldb`: This package helps Flask to connect to a MySQL DB. We have added this in requirements.txt.
2. `app.py` line 6-12: This adds configuration for the database so that the MySQL package can use it to connect to the database and create a MySQL instance.
3. Remove the dictionary and create a db.connection.cursor()(line 22) object which will help Flask to execute SQL query on the db(line 31).

We have also added one DB folder to the code. It has a init.sql file in it. The purpose of this is very simple: we have created a DB in the MySQL container using environment variables.

To authenticate users, we need to create a table in that database. So, either you can execute SQL commands in init.sql directly from the SQL shell or leverage Docker's docker-entrypoint-initdb.d folder.

docker-entrypoint-initdb.d explanation

Any SQL script present in docker-entrypoint-initdb.d folder is executed by Docker as soon as a container is up and running.

So, to leverage this feature, we will mount the db/init.sql file onto Docker's initdb.d/ folder using -v argument while running the container and do the following:

- Stop the existing MySQL container
- Remove it using the Docker rm command
- Type `docker run --name mysql -d -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=backend -e MYSQL_USER=testuser -e MYSQL_PASSWORD=admin123 -v <Path/to/Project/>/Docker/db/init.sql:/docker-entrypoint-initdb.d/init.sql mysql/mysql-server:5.7`

We have just added two new things in the command:

`-v <Path/to/Project/>/Docker/db/init.sql:/docker-entrypoint-initdb.d/init.sql` to mount the DB startup script and `-d` to run the container in the background.

Next, stop the app container, build a new image for our new code `docker build -t flask_app:2.0`, and run using `docker run --link "mysql:backenddb" -p 5000:5000 flask_app:1.0`.

```
CREATE TABLE IF NOT EXISTS `users` (
  `user_id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(255) DEFAULT NULL,
  `password` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`user_id`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=10001 ;

INSERT INTO `users` (`user_id`, `username`,  `password`) VALUES
(1, 'admin', 'admin123');
```

If everything goes well, you should be able to see the app and authenticate with `admin' and 'admin123' as username and password respectively.

If not, check out the troubleshooting lesson and try to troubleshoot.

In the next lesson, we will shortly summarize the steps we followed and see how to optimize these steps and use only one command to get this whole setup running using Docker-compose.