Docker Networking

Docker networking refers to the way Docker containers communicate with each other and with the outside world. When you run multiple Docker containers on a single host or across multiple hosts, they need to be able to communicate efficiently and securely. Docker provides several networking options to facilitate this communication:

Bridge network: This is the default network mode in Docker. In this mode, Docker creates a virtual bridge called **docker0**, and each container is assigned an IP address on this bridge network. Containers on the same bridge network can communicate with each other directly using these IP addresses.

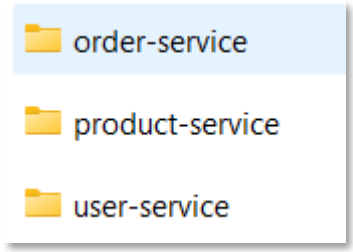
1. **Host network**: Here the containers share the network namespace with the Docker host, so they have direct access to the host's network interfaces. This can be useful for achieving better network performance, as there is no overhead from network address translation (NAT), but it also exposes containers directly to the host's network.
2. **Overlay network**: Overlay networks allow containers to communicate across multiple Docker hosts. They use VXLAN (Virtual Extensible LAN) technology to encapsulate and route network traffic between hosts. This is particularly useful for deploying distributed applications across a cluster of Docker hosts.
3. **Macvlan network**: Allows Docker containers to have their own MAC addresses and appear as separate physical devices on the network. This can be useful for scenarios where you want containers to be directly addressable from other devices on the network, without any NAT or port mapping.
4. **None network**: In this mode, Docker containers have no networking. This can be useful for certain special cases where you want to completely isolate a container from the network.

Docker networking also includes features for network security, such as firewalls and access control lists (ACLs), to control which containers can communicate with each other and with external networks. Additionally, Docker provides tools for monitoring and troubleshooting network connections between containers.

Networking Lab

In this lab, you will create a few isolated services (APIs) and create a network between them so they can call each other’s APIs

1- Create folder for the structure we need for three Micro services



2- Navigate to each folder and the run the following command to initialise a Node app:

```
npm init -y
```

3- Create an empty file in each folder called **Dockerfile**

4- Add a file in each folder called server.js
This file will hold the logic of your API for each service

5- You will now modify each **package.json** to grab the dependency on the express module and setup server.js as the main file.

```
{
  "name": "order-service",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "express": "^4.19.2"
  }
}
```

6- Copy the following text to the Dockerfile of the order-service

```
# Use Node.js LTS version as base image
FROM node:14

# Set working directory in the container
WORKDIR /app

# Copy package.json and package-lock.json to container
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy all local files to container
```

```
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Command to run the application
CMD ["node", "server.js"]
```

7- Add the same text to the Dockerfiles of the **product-service** and **user-service**

8- Copy the following text to the **order-service**'s server.js file

```
// order-service/server.js

const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

// Dummy data for orders
const orders = [
    { id: 1, productId: 1, quantity: 2 },
    { id: 2, productId: 2, quantity: 1 }
];

// API endpoint to get all orders
app.get('/orders', (req, res) => {
    res.json(orders);
});

// API endpoint to get a specific order by ID
app.get('/orders/:id', (req, res) => {
    const orderId = parseInt(req.params.id);
    const order = orders.find(o => o.id === orderId);
    if (order) {
        res.json(order);
    } else {
        res.status(404).json({ message: 'Order not found' });
    }
});

// Start the server
app.listen(PORT, () => {
    console.log(`Order service running on port ${PORT}`);
});
```

9- Copy the following text to the **product-service**'s server.js file

```
// product-service/server.js

const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

// Dummy data for products
const products = [
    { id: 1, name: 'Product 1', price: 10.99 },
    { id: 2, name: 'Product 2', price: 20.49 },
    { id: 3, name: 'Product 3', price: 15.99 }
];

// API endpoint to get all products
app.get('/products', (req, res) => {
    res.json(products);
});
```

```
// API endpoint to get a specific product by ID
app.get('/products/:id', (req, res) => {
    const productId = parseInt(req.params.id);
    const product = products.find(p => p.id === productId);
    if (product) {
        res.json(product);
    } else {
        res.status(404).json({ message: 'Product not found' });
    }
});

// Start the server
app.listen(PORT, () => {
    console.log(`Product service running on port ${PORT}`);
});
```

10- Finally, copy the following text to the **user-service**'s server.js file

```
// user-service/server.js

const express = require('express');
const app = express();
const PORT = process.env.PORT || 3000;

// Dummy data for users
const users = [
    { id: 1, username: 'user1', email: 'user1@example.com' },
    { id: 2, username: 'user2', email: 'user2@example.com' }
];

// API endpoint to get all users
app.get('/users', (req, res) => {
    res.json(users);
});

// API endpoint to get a specific user by ID
app.get('/users/:id', (req, res) => {
    const userId = parseInt(req.params.id);
    const user = users.find(u => u.id === userId);
    if (user) {
        res.json(user);
    } else {
        res.status(404).json({ message: 'User not found' });
    }
});

// Start the server
app.listen(PORT, () => {
    console.log(`User service running on port ${PORT}`);
});
```

11- Let's build our dockers and run them on a network by running the following command (start from the root folder, above the product-service folder)

```
docker network create ecommerce-network
```

12- Build and run the dockers

```
cd product-service
docker build -t product-service .

cd ..\order-service
docker build -t order-service .

cd ..\user-service
docker build -t user-service .
```

```
docker run -d --name product-service --network ecommerce-network
product-service

docker run -d --name order-service --network ecommerce-network
order-service

docker run -d --name user-service --network ecommerce-network
user-service
```

13- Now, start the dockers.

```
docker start product-service
docker start order-service
docker start user-service
```

13- This time when you test the products service, it will not work because the service does not expose itself out side of the ecommerce-network

```
curl http://product-service:3000/products
```

1- Let's test our network by calling a service from another:
Run an interactive shell Bash command in one of the services such as:

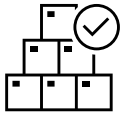
docker exec -it user-service bash

From the inside of the user-service, make a call to one of the other

services like:

```
curl http://product-service:3000/products
```

This time it will work because the services can communicate within the network.



Congratulations, you have successfully created a network between Dockers.

Time to experiment!

Consider adapting the application mentioned above for your own custom services, setting up a scenario that fits your specific needs.

You may create a service as part of the network with an external API end point like:

```
docker run -d -p 3000:3000 --name order-service --network ecommerce-network order-service
```

You may then call such a service like http://localhost:3000/orders

Tip: to make changes to order-service do:

- Stop the service: **`docker stop`** command
- Rebuild: **`docker build -t order-service .`**
- And run:
  `docker run -d -p 3000:3000 --name order-service --network ecommerce-network order-service`