pravinmishraaws /
Docker-Demo

Code Issues Pull requests Actions Projects Wiki Security

main Docker-Demo / 05_Docker_Networking.md

pravinmishraaws Update 05_Docker_Networking.md 8db8807 · 2 weeks ago

939 lines (722 loc) · 25.5 KB

Docker Networking

Networking is a crucial aspect of Docker, as it controls how containers communicate with each other and the outside world. This guide will:

- Explain the different Docker networking modes
- Demonstrate how to connect containers inside Docker networks
- Provide real-world examples
- Analyze command outputs to ensure clarity

1. Understanding Docker Networks

Docker provides multiple networking modes, each suited for different use cases.

Network Mode	Description
Bridge (Default)	Containers are isolated but can communicate using explicit port mappings.
Host	The container shares the host's network stack, removing network isolation.
None	The container has no network access, making it useful for security.

Network Mode	Description
Overlay	Used in multi-host Swarm setups for inter-container communication.
macvlan	A local scope network driver which is configured per-host

How to List Available Networks

Before diving into practical scenarios, it is important to check the existing Docker networks on your machine.

```
docker network ls
```

Expected Output:

```
NETWORK ID     NAME      DRIVER    SCOPE
7b369f1d82a3   bridge    bridge    local
44e59ebc37a7   host      host      local
4bb30e91a290   none      null      local
```

- **bridge**: Default network mode. Containers in this network can communicate using IP addresses.
- **host**: The container shares the host machine's network.
- **none**: The container has no network access.

Now, let's apply this knowledge in real-world scenarios.

Scenario 1: Running a Standalone Application

Problem Statement

A React application needs to be deployed using Docker and must be accessible from a web browser. Before running the application, we must ensure the necessary Docker image (Nginx) is available on the system.

Step 1: Check if the Nginx Image is Available Locally

Before pulling an image from Docker Hub, check whether it already exists on the local system using:

```
docker images
```

Expected Output (If the Image is Already Available)

```
REPOSITORY   TAG       IMAGE ID       CREATED        SIZE
nginx        latest    6b361163bff3   3 days ago     142MB
```

- `REPOSITORY` : The name of the image.
- `TAG` : The image version (e.g., `latest`).
- `IMAGE ID` : A unique identifier for the image.
- `CREATED` : When the image was created.
- `SIZE` : The storage size of the image.

If the **Nginx image is not listed**, proceed to the next step.

Step 2: Search for the Nginx Image on Docker Hub

To find available versions of Nginx on Docker Hub, use:

```
docker search nginx
```

Expected Output

```
NAME                              DESCRIPTION
STARS   OFFICIAL   AUTOMATED
nginx                             Official Nginx Docker Image
18542   [OK]
jwilder/nginx-proxy               Automated Nginx reverse proxy
2342
bitnami/nginx                     Bitnami Nginx container
1234
```

- The **official** Nginx image is marked with `[OK]` .
- Other versions (e.g., `bitnami/nginx`) are available but are maintained by different organizations.

Since we want the **official** version, we can now **download it**.

Step 3: Pull the Nginx Image (If Not Available Locally)

If the image is missing, download it from Docker Hub using:

```
docker pull nginx
```

Expected Output

```
Using default tag: latest
latest: Pulling from library/nginx
6ec8c9369e08: Pull complete
d695473f6229: Pull complete
Digest: sha256:XXXXXXXXXXXX
Status: Downloaded newer image for nginx:latest
```

- The image layers are downloaded and stored locally.
- The `Status: Downloaded newer image` confirms the successful download.

Step 4: Check If Any Containers Are Running

Before proceeding, check if any containers are already running:

```
docker ps
```

Expected Output (If No Containers Are Running)

```
CONTAINER ID   IMAGE   COMMAND   CREATED   STATUS   PORTS   NAMES
```

If no containers are listed, no applications are currently running.

If any containers **are running**, stop them if necessary using:

```
docker stop <container_id>
```

Step 5: Run the Nginx Container

```
docker run -d -p 80:80 --name myweb nginx
```

Command Breakdown

- `docker run` : Starts a new container.
- `-d` : Runs the container in detached mode (in the background).
- `-p 80:80` : Maps port 8080 on the host machine to port 80 inside the container. This allows the application to be accessed via port 8080.
- `--name myweb` : Assigns the container a readable name for easy management.
- `nginx` : Specifies the Nginx image, which serves static files.

Step 6: Verify the Container is Running

```
docker ps
```

Expected Output

```
CONTAINER ID   IMAGE    COMMAND                  PORTS                  NAMES
c1d5f7e2a15b   nginx    "/docker-entrypoint.…"   0.0.0.0:80->80/tcp     myweb
```

- The container is now running and accessible.

Step 7: Access the Application

Open a web browser and go to:

```
http://PublicIP
```

You should see the Nginx welcome page.

Key Takeaways

- The `docker images` command helps verify if an image is available locally.
- The `docker search` command allows searching for images on Docker Hub.
- The `docker pull` command downloads an image when needed.

- Running `docker ps` ensures no conflicting containers are running before deployment.
- Port mapping enables external access to the containerized application.

Scenario 2: Two Containers Communicating (Microservices)

Problem Statement

A frontend application needs to fetch data from a backend API inside Docker. Using `localhost` will not work because each container runs in its own isolated environment.

Solution

A **custom bridge network** will be created so that containers can communicate **using container names instead of IP addresses**.

Step 1: Cleanup Previous Containers and Images

Before setting up the new scenario, stop and remove any containers and images from the **previous scenario**.

Stop and Remove the Running Container

Check if any containers are running:

```
docker ps
```

Expected output (if any containers are running):

```
CONTAINER ID   IMAGE     COMMAND                  PORTS                  NAMES
c1d5f7e2a15b   nginx     "/docker-entrypoint.…"   0.0.0.0:8080->80/tcp   myweb
```

If the `myweb` container is running, stop it:

```
docker stop myweb
```

Remove the container:

```
docker rm myweb
```

Remove Unused Images

Check for unused images:

```
docker images
```

If the Nginx image is no longer needed, remove it:

```
docker rmi nginx
```

At this point, the system is clean and ready for the new scenario.

Step 2: Create a Custom Network

Before running any containers, create a dedicated network where the backend and frontend will communicate.

```
docker network create mynetwork
```

Command Breakdown

- `docker network create mynetwork` : Creates a new bridge network named `mynetwork` . Containers connected to this network can communicate using their names instead of IP addresses.

Verify that the network was created successfully:

```
docker network ls
```

Expected output:

```
NETWORK ID     NAME        DRIVER    SCOPE
7b369f1d82a3   bridge      bridge    local
44e59ebc37a7   host        host      local
4bb30e91a290   none        null      local
a9f58b94c8a0   mynetwork   bridge    local
```

The new `mynetwork` should be listed.

Step 3: Run the Frontend Application

A: Create a Dockerfile for the Frontend

Instead of running Frontend commands manually, a **Dockerfile** will be used to build a containerized Frontend service.

Updated Dockerfile

```
# Use the official Node.js 18 image
FROM node:18

# Set the working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json if they exist (optional if us
COPY package*.json ./

# Initialize a new Node.js app and install Express
RUN npm init -y && npm install express

# Create index.js with basic Express code running on port 80
RUN echo "const express = require('express'); \
const app = express(); \
app.get('/', (req, res) => res.send('Hello from Frontend')); \
const PORT = 80; \
app.listen(PORT, () => console.log('Frontend running on port ' + PORT))

# Expose port 80 to the host
EXPOSE 80

# Run the application
CMD ["node", "index.js"]
```

B: Build and Run the Frontend Container

After creating the `Dockerfile`, the next steps are:

Build the Docker Image

```
docker build -t frontend-app .
```

Run the Backend Container in the Custom Network

```
docker run -d --name frontend --network mynetwork -p 80:80 frontend-app
```

Command Breakdown

- `docker build -t frontend-app .` : Creates a Docker image named `frontend-app`.
- `docker run -d --name frontend` : Runs the Frontend container in detached mode with the name `frontend`.
- `--network mynetwork` : Ensures the Frontend is connected to `mynetwork`.
- `-p 80:80` : Maps **container port 80** to **host port 80**.

C: Verify Frontend is Running

Check the running containers:

```
docker ps
```

Expected output:

```
CONTAINER ID   IMAGE         COMMAND                  PORTS                  NAMES
b1d5a7c8e9f2   frontend-app   "node index.js"          0.0.0.0:80->80/tcp     frontend
```

Test the Backend from the Host

Run the following command:

```
curl http://<publicIP>
```

Expected output:

```
Hello from Frontend
```

This confirms that:

- The **Frontend container is running on port 80**.
- It is accessible from the **host machine**.

Step 4: Run the Backend

Now, deploy an Nginx container as the Backend.

```
docker run -d --name backend --network mynetwork nginx
```

Command Breakdown

- `docker run -d` : Runs the container in detached mode.
- `--name backend` : Assigns the container the name `backend` .
- `--network mynetwork` : Connects the container to `mynetwork` .
- `nginx` : Uses Nginx as the backend service.

Verify that the backend is running:

```
docker ps
```

Expected output:

```
CONTAINER ID   IMAGE    COMMAND                  PORTS
NAMES
d7cfa6a9b8e2   node:18  "bash -c 'npm init …"   0.0.0.0:80->80/tcp
frontend
a8d2c9f8e6a7   nginx    "/docker-entrypoint.…"  80/tcp
backend
```

Both `backend` and `frontend` should be listed.

Step 5: Verify Network Connectivity

Inspect the custom network:

```
docker network inspect mynetwork
```

Expected Output

```
"Containers": {
    "backend": {
        "Name": "backend",
        "IPv4Address": "192.168.1.2/24"
    },
    "frontend": {
        "Name": "frontend",
        "IPv4Address": "192.168.1.3/24"
    }
}
```

This confirms that both containers are assigned internal IPs and are part of the same network.

Step 6: Test Communication Between Frontend and Backend

Option 1: Test Manually

Now, check if the `frontend` container can access the `backend` API using its **container name**.

```
docker exec -it frontend apt update && docker exec -it frontend apt ins
docker exec -it frontend curl backend
```

Command Breakdown

- `docker exec -it frontend` : Runs a command inside the `frontend` container.
- `apt update` : Updates the package lists (since the Nginx image does not include `curl` by default).
- `apt install curl -y` : Installs `curl` inside the frontend container.

- `curl backend:3000` : Tries to access the backend API.

Expected Output

```
Welcome to nginx!
If you see this page, the nginx web server is successfully installed
and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.
```

If this output appears, it confirms that:

- The `backend` container is running properly.
- The `frontend` container can resolve `backend` by name.
- The **custom bridge network is working correctly**.

Option 2: Enter the Frontend Container

Instead of running `docker exec` commands from the host, you can enter the `frontend` container interactively and then test connectivity.

Step 1: Access the Frontend Container

```
docker exec -it frontend /bin/sh
```

This opens an interactive shell inside the container.

Step 2: Install Curl Inside the Container

Since the **default Nginx container does not have curl installed**, update the package lists and install curl:

```
apt update && apt install curl -y
```

Step 3: Test Backend API from Inside the Frontend Container

```
curl backend
```

Expected output:

```
Welcome to nginx!
If you see this page, the nginx web server is successfully installed
and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.
```

Step 4: Exit the Frontend Container

After testing, exit the container by running:

```
exit
```

Scenario 3: Multi-Tier Architecture (Frontend, Backend, Database)

Problem Statement

A complete web application consists of three components:

- **Frontend (React)**
- **Backend (Node.js)**
- **Database (MongoDB)**

Each service must communicate with others while maintaining **security and isolation**:

- The **database should be hidden** from external access.
- The **backend should access the database** but should not be exposed to the internet.
- The **frontend should only talk to the backend** but should not directly access the database.

Solution

We will use **multiple Docker networks** to control communication:

1. **Backend network**: Connects the backend and the database.
2. **Frontend network**: Connects the frontend and the backend.

Step 1: Clean Up Old Resources

Before setting up the new architecture, stop and remove all running containers, delete networks, and ensure a fresh start.

Stop and Remove Running Containers

```
docker ps -q | xargs -r docker stop
docker ps -aq | xargs -r docker rm
```

- `docker ps -q` lists all running container IDs.
- `docker ps -aq` lists all container IDs (including stopped ones).
- `xargs -r docker stop` stops all running containers.
- `xargs -r docker rm` removes all containers.

Remove Unused Images

```
docker images -q | xargs -r docker rmi -f
```

- Removes all unused images.

Now, the system is clean and ready for setup.

Step 2: Create the Required Networks

```
docker network create backend-network
docker network create frontend-network
```

- `backend-network` : Used by the backend and database.

- `frontend-network` : Used by the frontend and backend.

Step 3: Set Up the Folder Structure

Create the project structure:

```
mkdir -p multi-tier-app/{frontend,backend,database}
```

Folder structure:

```
multi-tier-app/
|-- frontend/
|   └── Dockerfile
|-- backend/
|   └── Dockerfile
|-- database/
|   └── Dockerfile
```

Now, create **Dockerfiles** for each layer.

Step 4: Create the Database Layer (MongoDB)

Inside the `multi-tier-app/database/` folder, create a **Dockerfile**:

```
cd multi-tier-app/database
touch Dockerfile
```

Dockerfile for MongoDB

```
# Use the official MongoDB image
FROM mongo:latest

# Expose MongoDB's default port
EXPOSE 27017
```

Build and Run the MongoDB Container

```
docker build -t my-mongo ./multi-tier-app/database
docker run -d --name db --network backend-network my-mongo
```

- The database **only exists within the backend network** and is not exposed to the internet.

Step 5: Create the Backend Layer

Inside the `multi-tier-app/backend/` folder, create a **Dockerfile**:

```
cd ../backend
touch Dockerfile
```

Dockerfile for Backend (Node.js)

```
# Use the official Node.js image
FROM node:18

# Set working directory
WORKDIR /app

# Copy package.json and install dependencies
COPY package*.json ./
RUN npm init -y && npm install express mongoose cors body-parser

# Copy application code
COPY . .

# Create index.js with Express API and MongoDB connection
RUN echo "const express = require('express'); \
const mongoose = require('mongoose'); \
const app = express(); \
const PORT = 80; \
mongoose.connect('mongodb://db:27017/mydb', { useNewUrlParser: true, use
.then(() => console.log('Connected to MongoDB')) \
.catch(err => console.log('MongoDB connection error:', err)); \
app.get('/', (req, res) => res.send('Hello from Backend')); \
app.listen(PORT, () => console.log('Backend running on port ' + PORT));'

# Expose port 80 for communication
EXPOSE 80
```

```
# Run the application
CMD ["node", "index.js"]
```

Build and Run the Backend Container

```
docker build -t backend-app .
docker run -d --name api --network backend-network backend-app
```

- The backend is only connected to the **backend network**.
- The backend can access MongoDB using `mongodb://db:27017/mydb`.

Step 6: Create the Frontend Layer

Inside the `multi-tier-app/frontend/` folder, create a **Dockerfile**:

```
cd ../frontend
touch Dockerfile
```

Dockerfile for Frontend (React with Nginx)

```
# Use the official lightweight Nginx image
FROM nginx:alpine

# Expose port 80 to allow incoming traffic
EXPOSE 80

# Start Nginx in the foreground
CMD ["nginx", "-g", "daemon off;"]
```

Build and Run the Frontend Container

```
docker build -t frontend-app .
docker run -d --name ui --network frontend-network -p 80:80 frontend-app
```

- The frontend runs on port 80 and is exposed to port 80.

Step 7: Connect Backend and Frontend Networks

Since the backend needs to communicate with both **MongoDB** and **Frontend**, it must be connected to both networks.

```
docker network connect frontend-network api
```

- The **frontend can reach the backend**.
- The **backend can still access MongoDB** but remains **hidden from the frontend**.

Step 8: Verify Network Connectivity

Check Network Connections

To verify that each service is correctly attached to the appropriate networks, inspect the networks:

```
docker network inspect backend-network
docker network inspect frontend-network
```

Expected output for `backend-network` :

```
"Containers": {
    "db": {
        "Name": "db",
        "IPv4Address": "192.168.1.2/24"
    },
    "api": {
        "Name": "api",
        "IPv4Address": "192.168.1.3/24"
    }
}
```

Expected output for `frontend-network` :

```
"Containers": {
    "api": {
        "Name": "api",
        "IPv4Address": "192.168.2.2/24"
```

```
        },
        "ui": {
            "Name": "ui",
            "IPv4Address": "192.168.2.3/24"
        }
    }
```

Test Frontend from the Host

Since the frontend is exposed to port `80`, it should be accessible via the public IP.

```
curl http://PublicIP
```

Expected output:

```
Welcome to nginx!
If you see this page, the nginx web server is successfully installed
and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.
```

This confirms that:

- The frontend **is running properly**.
- Nginx is **serving the application**.

Test Frontend and Backend Communication

Now, verify if the **frontend can communicate with the backend**.

1. Enter the **frontend container**:

   ```
   docker exec -it ui /bin/sh
   ```

2. Inside the **frontend container**, run:

   ```
   curl api
   ```

Expected output:

```
Hello from Backend
```

This confirms:

- The **frontend successfully connects to the backend**.
- The **backend is responding** to requests.

Test Backend and Database Communication

Now, verify that the **backend can communicate with MongoDB**.

1. Enter the **backend container**:

```
docker exec -it api /bin/sh
```

Step 1: Access the Backend Container

First, enter the **backend container** (`api`):

```
docker exec -it api /bin/sh
```

Install MongoDB Shell (`mongosh`) Inside the Backend Container**


```
curl -fsSL https://www.mongodb.org/static/pgp/server-6.0.asc | gpg --dea

echo "deb [signed-by=/usr/share/keyrings/mongodb-server-6.0.gpg] http:/,

apt-get update
apt-get install -y mongodb-org
```

```
systemctl start mongod
systemctl enable mongod
```

2. Inside the **backend container**, run the MongoDB shell:

```
mongosh --host db --port 27017
```

Expected output:

```
Current MongoDB server version: 5.x.x
Connecting to: mongodb://db:27017/
```

3. Create a test database and collection:

```
use testdb
db.users.insertOne({ name: "Alice", age: 25 })
```

Expected output:

```
{ acknowledged: true, insertedId: ObjectId("...")
```

4. Retrieve data to confirm the insertion:

```
db.users.find()
```

Expected output:

```
{ "_id": ObjectId("..."), "name": "Alice", "age": 25 }
```

5. Exit the MongoDB shell:

```
exit
```

6. Exit the backend container:

```
exit
```

This confirms:

- The **backend can connect to the database**.

- MongoDB **is working correctly**.
- Data can be **written and retrieved** from the database.

Scenario 4: Using Host Mode for High-Performance Applications

Problem Statement

A high-performance application requires direct access to the host's network to reduce latency and avoid network overhead. Docker's default bridge network introduces **Network Address Translation (NAT)**, which can add delays.

To eliminate this overhead, we can use **host networking**, where the container shares the host's network stack.

Solution: Use Host Networking Mode

By using **Docker's host networking mode**, the container will:

- Directly use the **host's IP address and network interfaces**.
- Eliminate the need for **NAT (Network Address Translation)**.
- Avoid explicit port mappings since the container runs as if it were a native process on the host.

Step 1: Clean Up Old Containers (Optional)

Before running a new container in **host mode**, ensure no conflicting containers are running.

Stop and Remove Any Running Containers

```
docker ps -q | xargs -r docker stop
docker ps -aq | xargs -r docker rm
```

Command Breakdown

- `docker ps -q` : Lists all running container IDs.
- `xargs -r docker stop` : Stops all running containers.
- `docker ps -aq` : Lists all container IDs (including stopped ones).
- `xargs -r docker rm` : Removes all containers.

Step 2: Run an Application in Host Network Mode

Deploy **Nginx**, a lightweight web server, using host networking.

```
docker run -d --network host --name fastapp nginx
```

Command Breakdown

- `docker run -d` : Runs the container in detached mode.
- `--network host` : Uses **host networking mode**, allowing the container to directly use the **host's network stack**.
- `--name fastapp` : Assigns the container a name for easier management.
- `nginx` : Uses the official Nginx web server image.

Step 3: Verify the Running Container

Since host networking **does not display exposed ports in `docker ps`**, check running containers:

```
docker ps
```

Expected output:

```
CONTAINER ID   IMAGE    COMMAND                  STATUS        NAMES
e3d7c6b2a8e1   nginx    "/docker-entrypoint.…"   Up 10 min
fastapp
```

Unlike **bridge mode**, no ports are listed because the container directly shares the host's networking stack.

Step 4: Access the Application

Since Nginx runs on **port 80** by default, access it using:

```
curl http://localhost
```

Expected Output

```
<!DOCTYPE html>
<html>
<head><title>Welcome to nginx!</title></head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
installed and working.</p>
</body>
</html>
```

Alternatively, open a web browser and go to:

```
http://PubliIP
```

The default Nginx welcome page should be visible.

Step 5: Verify the Network Mode

To confirm that the container is **running in host mode**, inspect it:

```
docker inspect fastapp | grep '"NetworkMode"'
```

Expected Output

```
"NetworkMode": "host",
```

This confirms that **Docker is not using bridge mode or any other virtualized network**.

Step 6: Stop and Remove the Container (Cleanup)

If you no longer need the container, remove it:

```
docker stop fastapp
docker rm fastapp
```

Key Takeaways

Feature	Bridge Mode (`--network bridge`)	Host Mode (`--network host`)
Container gets its own IP	Yes	No (Uses host's IP)
NAT (Network Address Translation)	Yes	No
Explicit Port Mapping (`-p`)	Required	Not needed
Best for	General applications	Low-latency applications

When to Use Host Mode

- Performance-critical workloads such as real-time data processing and low-latency APIs.
- Monitoring and networking tools that need access to host-level network traffic.
- Applications that must bind to a specific port without remapping, such as VoIP or gaming servers.

When Not to Use Host Mode

- If port conflicts may occur since the container shares the same ports as the host.
- If multiple containers need to run on the same ports.
- When security isolation is required, as containers in host mode have direct access to the host system.