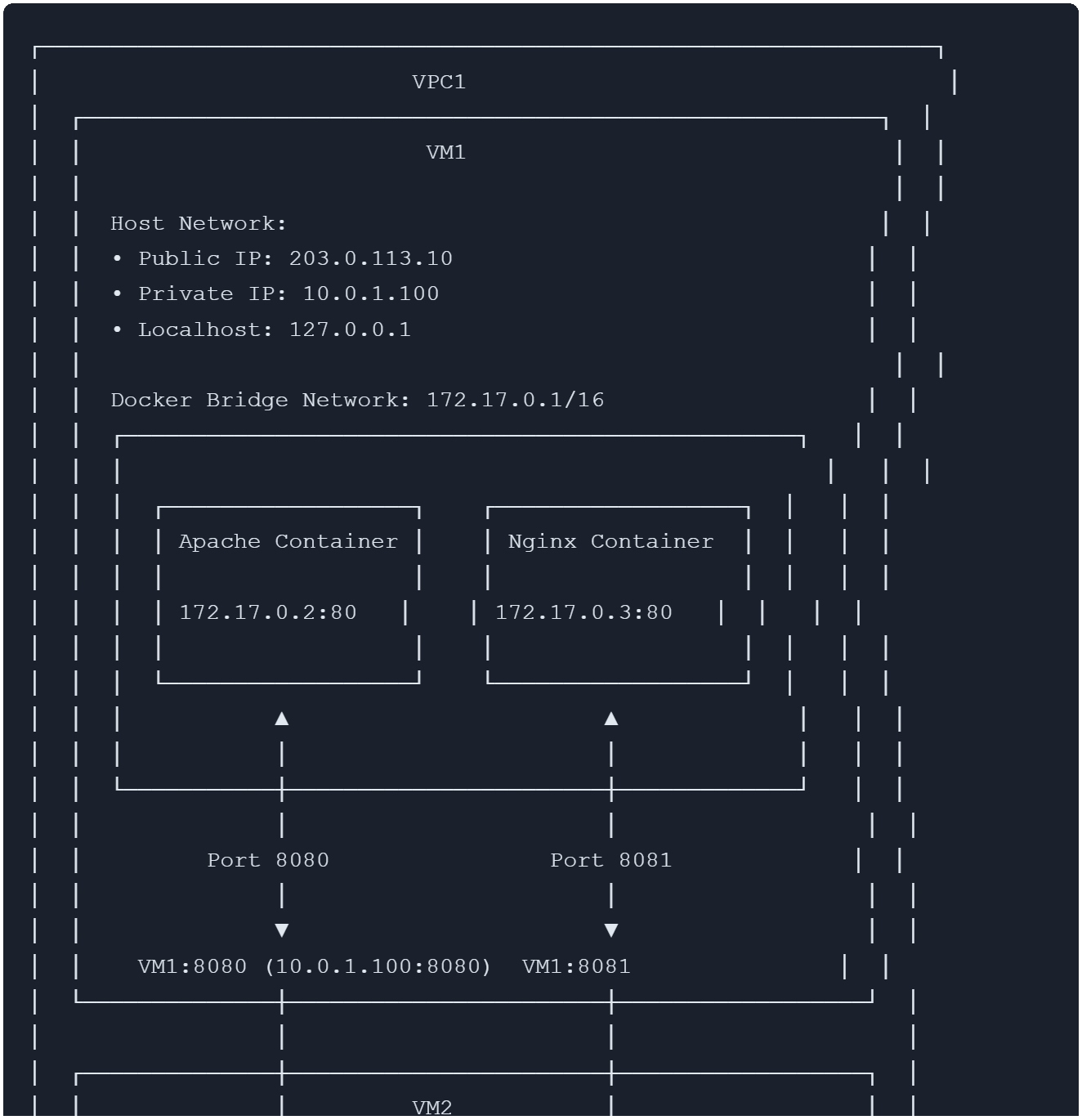
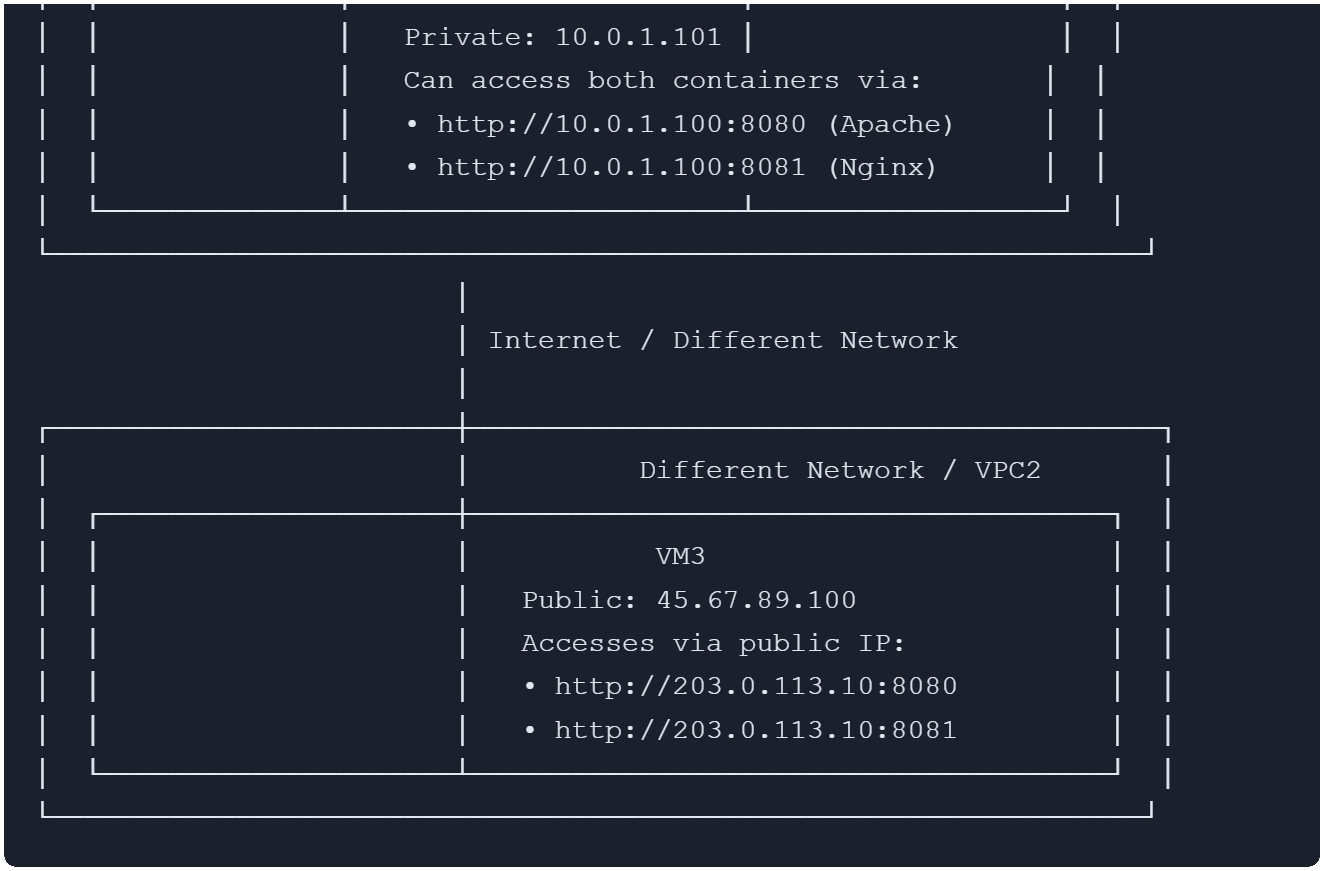
Docker Container Networking: Multi-VM Scenarios

Scenario Overview

This guide demonstrates how to expose Docker containers running on VM1 to other VMs within the same VPC and across different networks.

Network Architecture

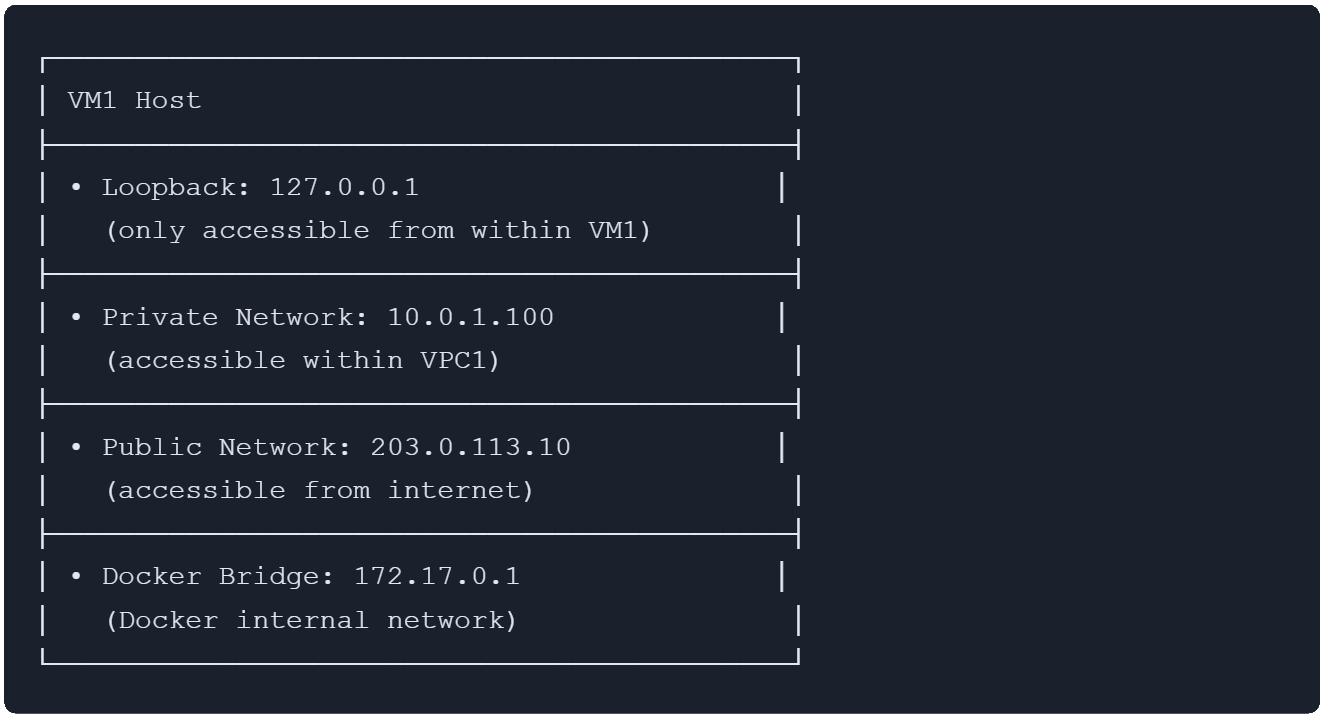




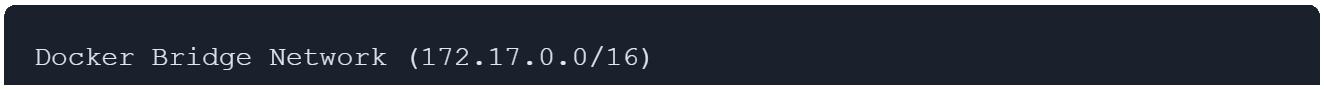
Understanding the Network Layers

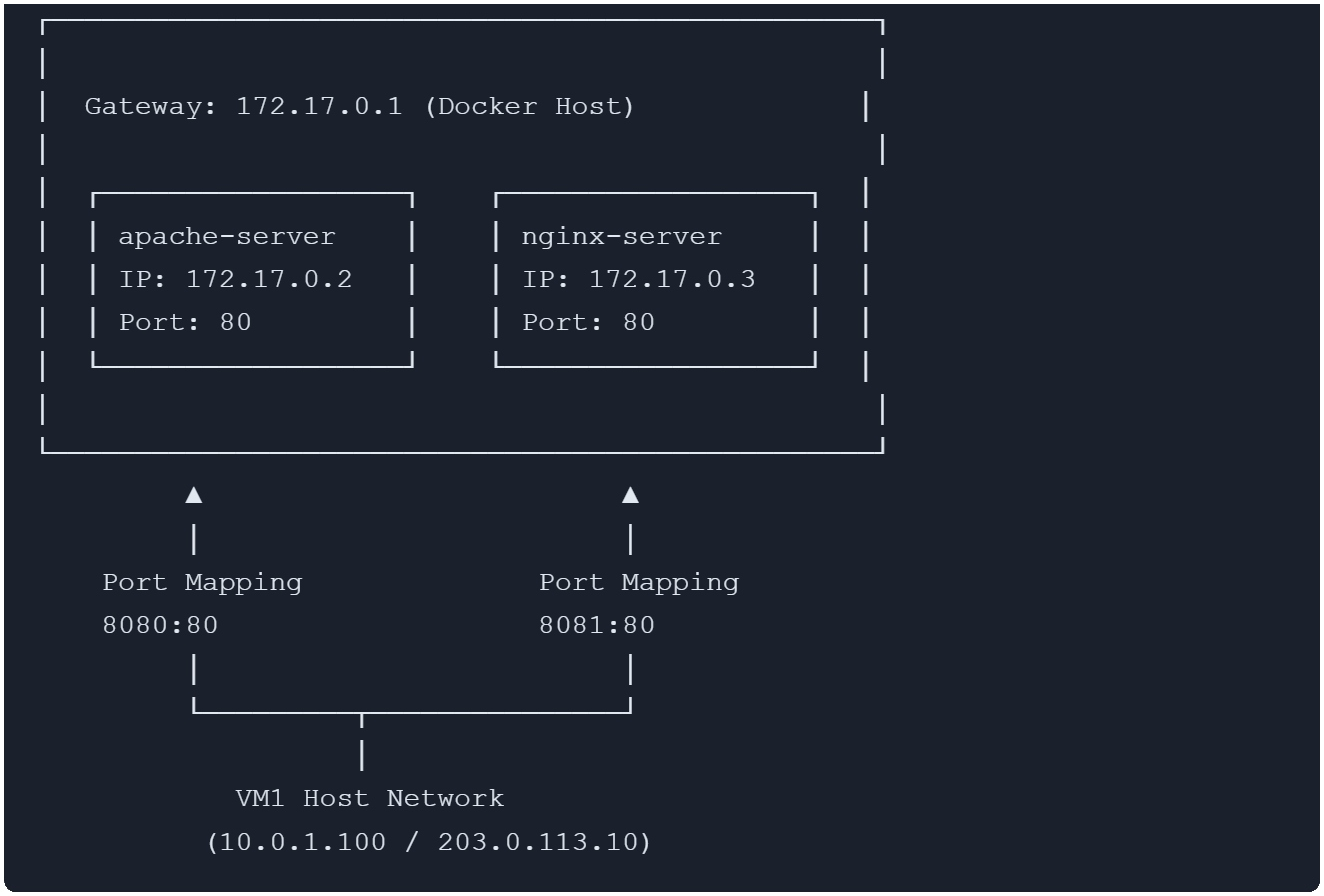
1. VM1 Network Interfaces

VM1 has three network layers:



2. Docker Bridge Network





Step-by-Step Setup

Step 1: Create Docker Containers on VM1

Apache Container Setup

```
# Run Apache container with port mapping
docker run -d \
  --name apache-server \
  -p 8080:80 \
  httpd:latest

# Verify container is running
docker ps

# Output:
# CONTAINER ID   IMAGE          COMMAND              STATUS         PORTS
# a1b2c3d4e5f6   httpd:latest   "httpd-foreground"   Up 5 seconds   0.0.0.0:80
```

What happens:

- Container gets internal IP: `172.17.0.2`
- Container listens on port `80` internally
- Port `8080` on VM1's host network maps to container port `80`

- Port `8080` on VM1's host network maps to container port `80`
- Traffic flow: `VM1:8080 → Docker Bridge → Container:80`

Nginx Container Setup

```
# Run Nginx container with port mapping
docker run -d \
  --name nginx-server \
  -p 8081:80 \
  nginx:latest

# Verify container is running
docker ps

# Output:
# CONTAINER ID   IMAGE          COMMAND                  STATUS         PORTS
# b2c3d4e5f6a7   nginx:latest   "/docker-entrypoint..."  Up 3 seconds   0.0.0.
# a1b2c3d4e5f6   httpd:latest   "httpd-foreground"       Up 1 minute    0.0.0.
```

What happens:

- Container gets internal IP: `172.17.0.3`
- Container listens on port `80` internally
- Port `8081` on VM1's host network maps to container port `80`
- Traffic flow: `VM1:8081 → Docker Bridge → Container:80`

Step 2: Verify Container Network Details

```
# Check container IP addresses
docker inspect apache-server -f '{{.NetworkSettings.IPAddress}}'
# Output: 172.17.0.2

docker inspect nginx-server -f '{{.NetworkSettings.IPAddress}}'
# Output: 172.17.0.3

# Check port mappings
docker port apache-server
# Output: 80/tcp -> 0.0.0.0:8080

docker port nginx-server
# Output: 80/tcp -> 0.0.0.0:8081
```

Port Mapping Explained

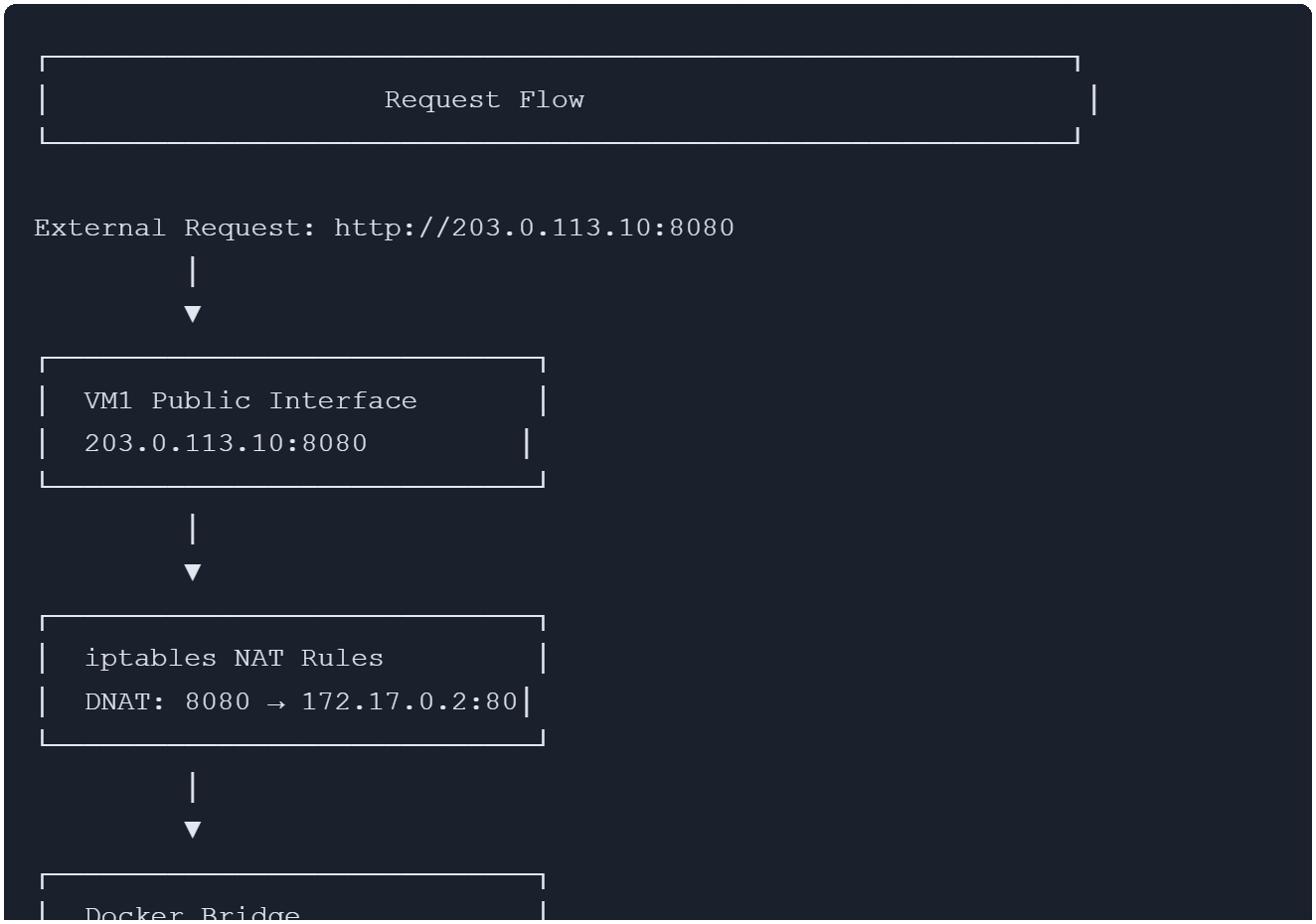
Understanding -p Flag

```
-p [HOST_IP:]HOST_PORT:CONTAINER_PORT[/PROTOCOL]
```

Examples:

```
# 1. Map port 8080 on all host interfaces to container port 80
-p 8080:80

# 2. Map to specific host interface (private network only)
-p 10.0.1.100:8080:80

# 3. Map to localhost only (only accessible from VM1)
-p 127.0.0.1:8080:80

# 4. Specify protocol
-p 8080:80/tcp
-p 53:53/udp

# 5. Map multiple ports
-p 8080:80 -p 8443:443
```

Port Mapping Flow Diagram



Accessing Containers from Different Sources

Scenario 1: Access from VM1 (Localhost)

```
# On VM1, test containers locally

# Apache - Multiple ways to access:
curl http://localhost:8080
curl http://127.0.0.1:8080
curl http://10.0.1.100:8080
curl http://172.17.0.2:80        # Direct container access

# Nginx
curl http://localhost:8081
curl http://127.0.0.1:8081
curl http://10.0.1.100:8081
curl http://172.17.0.3:80        # Direct container access

# Output examples:
# <html><body><h1>It works!</h1></body></html>  (Apache)
# <!DOCTYPE html><html>...Welcome to nginx...</html>  (Nginx)
```

Access Matrix from VM1:

Method	Apache	Nginx	Notes
`127.0.0.1:8080`	✅	N/A	Localhost only
`127.0.0.1:8081`	N/A	✅	Localhost only

`10.0.1.100:8080`	✅	N/A	Private IP
`10.0.1.100:8081`	N/A	✅	Private IP
`172.17.0.2:80`	✅	N/A	Direct container IP
`172.17.0.3:80`	N/A	✅	Direct container IP

Scenario 2: Access from VM2 (Same VPC)

VM2 is in the same VPC1 with private IP `10.0.1.101`

```
# On VM2, access containers via VM1's private IP

# Apache
curl http://10.0.1.100:8080
wget http://10.0.1.100:8080 -O apache-response.html

# Nginx
curl http://10.0.1.100:8081
wget http://10.0.1.100:8081 -O nginx-response.html

# Can also use VM1's public IP (if allowed by security groups)
curl http://203.0.113.10:8080
curl http://203.0.113.10:8081
```

Network Path:

```
VM2 (10.0.1.101)
       |
       | Internal VPC routing
       ▼
VM1 (10.0.1.100:8080)
       |
       | Port mapping
       ▼
Docker Bridge (172.17.0.1)
       |
       ▼
Apache Container (172.17.0.2:80)
```

Why VM2 CANNOT access 172.17.0.x directly:

- Docker bridge network is isolated to VM1
- Only VM1 can route to 172.17.0.0/16

- External VMs must use VM1's host ports

Scenario 3: Access from VM3 (Different Network/VPC)

VM3 is in a different network (VPC2) with IP `45.67.89.100`

```
# On VM3, must access via VM1's public IP

# Prerequisite: VM1 must have:
# - Public IP assigned: 203.0.113.10
# - Security group/firewall rules allowing inbound traffic on ports 8080, 808

# Apache
curl http://203.0.113.10:8080

# Nginx
curl http://203.0.113.10:8081

# Using telnet to test connectivity
telnet 203.0.113.10 8080
telnet 203.0.113.10 8081

# Using browser
# http://203.0.113.10:8080
# http://203.0.113.10:8081
```

Network Path:

```
VM3 (45.67.89.100) - Different VPC/Network
       |
       | Internet / VPN / Peering
       ▼
Internet Gateway / Router
       |
       ▼
VM1 Public IP (203.0.113.10:8080)
       |
       | NAT / Port forwarding
       ▼
VM1 Private Interface (10.0.1.100:8080)
       |
       | Docker port mapping
       ▼
Docker Bridge (172.17.0.1)
       |
       ▼
```

```
Apache Container (172.17.0.2:80)
```

Security Configuration

Firewall Rules Required

On VM1 (Host Level)

```
# Allow incoming traffic on mapped ports (if using iptables)
sudo iptables -A INPUT -p tcp --dport 8080 -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 8081 -j ACCEPT

# Or using firewalld (CentOS/RHEL)
sudo firewall-cmd --permanent --add-port=8080/tcp
sudo firewall-cmd --permanent --add-port=8081/tcp
sudo firewall-cmd --reload

# Or using ufw (Ubuntu)
sudo ufw allow 8080/tcp
sudo ufw allow 8081/tcp
```

Cloud Provider Security Groups

```
# AWS Security Group Example
Inbound Rules:
  - Type: Custom TCP
    Protocol: TCP
    Port: 8080
    Source: 10.0.1.0/24        # Allow from VPC (VM2)

  - Type: Custom TCP
    Protocol: TCP
    Port: 8081
    Source: 10.0.1.0/24        # Allow from VPC (VM2)

  - Type: Custom TCP
    Protocol: TCP
    Port: 8080
    Source: 45.67.89.100/32    # Allow from VM3

  - Type: Custom TCP
    Protocol: TCP
    Port: 8081
    Source: 45.67.89.100/32    # Allow from VM3
```

Complete Docker Commands Reference

Running Containers with Different Port Mappings

```
# 1. Apache - accessible from anywhere (0.0.0.0)
docker run -d \
  --name apache-server \
  -p 8080:80 \
  --restart unless-stopped \
  httpd:latest

# 2. Nginx - accessible from anywhere
docker run -d \
  --name nginx-server \
  -p 8081:80 \
  --restart unless-stopped \
  nginx:latest

# 3. Apache - accessible only from private network
docker run -d \
  --name apache-private \
  -p 10.0.1.100:8080:80 \
  httpd:latest

# 4. Nginx - accessible only from localhost
docker run -d \
  --name nginx-local \
  -p 127.0.0.1:8081:80 \
  nginx:latest

# 5. Multiple ports for single container
docker run -d \
  --name web-server \
  -p 8080:80 \
  -p 8443:443 \
  nginx:latest
```

Verify and Test

```
# Check running containers
docker ps --format "table {{.Names}}\t{{.Image}}\t{{.Ports}}"

# Output:
# NAMES           IMAGE          PORTS
# nginx-server    nginx:latest   0.0.0.0:8081->80/tcp
# apache-server   httpd:latest   0.0.0.0:8080->80/tcp
```

```
# Test from VM1
curl -I http://localhost:8080
curl -I http://localhost:8081

# View real-time logs
docker logs -f apache-server
docker logs -f nginx-server

# Check container stats
docker stats apache-server nginx-server

# Output:
# CONTAINER       CPU %   MEM USAGE / LIMIT     NET I/O
# apache-server   0.01%   11.5MiB / 1.952GiB   1.5kB / 2.1kB
# nginx-server    0.02%   8.2MiB / 1.952GiB    1.2kB / 1.8kB
```

Troubleshooting

Common Issues and Solutions

1. Cannot Access from VM2/VM3

```
# Problem: Connection timeout or refused

# Check 1: Verify containers are running
docker ps

# Check 2: Verify port mapping
docker port apache-server
docker port nginx-server

# Check 3: Test from VM1 first
curl http://localhost:8080
curl http://localhost:8081

# Check 4: Verify firewall (on VM1)
sudo iptables -L -n | grep 8080
sudo iptables -L -n | grep 8081

# Check 5: Test if port is listening
netstat -tulpn | grep 8080
netstat -tulpn | grep 8081

# Check 6: View iptables NAT rules
sudo iptables -t nat -L -n | grep 8080
```

2. Port Already in Use

```
# Error: "port is already allocated"

# Find what's using the port
sudo lsof -i :8080
sudo netstat -tulpn | grep 8080

# Kill the process or use different port
docker run -d -p 8082:80 httpd:latest
```

3. Container Not Reachable via Direct IP

```
# Problem: Cannot access 172.17.0.2 from VM2

# This is EXPECTED behavior
# Docker bridge network is isolated to VM1 host
# Solution: Use port mapping and VM1's host IP
curl http://10.0.1.100:8080
```

Advanced Networking Options

Using Custom Bridge Network

```
# Create custom bridge network
docker network create \
  --driver bridge \
  --subnet 172.18.0.0/16 \
  --gateway 172.18.0.1 \
  custom-net

# Run containers on custom network
docker run -d \
  --name apache-custom \
  --network custom-net \
  -p 9080:80 \
  httpd:latest

docker run -d \
  --name nginx-custom \
  --network custom-net \
  -p 9081:80 \
  nginx:latest
```

```
# Containers can communicate by name
docker exec apache-custom curl http://nginx-custom:80
```

Using Host Network Mode

```
# Container uses host's network directly
# No isolation, no port mapping needed
docker run -d \
  --name apache-host \
  --network host \
  httpd:latest

# Container binds directly to host's port 80
# Access via: http://10.0.1.100:80
# Note: Less secure, only use when necessary
```

Summary Table

Access Methods Comparison

Source	Target	URL	Requirements
VM1	Apache	`http://127.0.0.1:8080`	Local only
VM1	Nginx	`http://127.0.0.1:8081`	Local only
VM1	Apache	`http://172.17.0.2:80`	Direct container access
VM2 (Same VPC)	Apache	`http://10.0.1.100:8080`	VPC routing enabled
VM2 (Same VPC)	Nginx	`http://10.0.1.100:8081`	VPC routing enabled
VM3 (Different VPC)	Apache	`http://203.0.113.10:8080`	Public IP + Security rules
VM3 (Different VPC)	Nginx	`http://203.0.113.10:8081`	Public IP + Security rules

Port Mapping Patterns

Pattern	Usage	Example	Security Level
`-p 8080:80`	All interfaces	`0.0.0.0:8080→80`	Low - Publicly accessible

`-p 10.0.1.100:8080:80`	Specific private IP	Private network only	Medium - VPC only
`-p 127.0.0.1:8080:80`	Localhost only	Local access only	High - Host only
`--network host`	Host network	No isolation	Low - Direct host access

Complete Working Example

```
# Complete setup script for VM1

#!/bin/bash

# Step 1: Pull images
docker pull httpd:latest
docker pull nginx:latest

# Step 2: Create custom network
docker network create app-network

# Step 3: Run Apache
docker run -d \
  --name apache-server \
  --network app-network \
  -p 8080:80 \
  --restart unless-stopped \
  -e APACHE_LOG_LEVEL=info \
  httpd:latest

# Step 4: Run Nginx
docker run -d \
  --name nginx-server \
  --network app-network \
  -p 8081:80 \
  --restart unless-stopped \
  nginx:latest

# Step 5: Verify
echo "Waiting for containers to start..."
sleep 5

echo "Testing Apache..."
curl -I http://localhost:8080

echo "Testing Nginx..."
```

```
curl -I http://localhost:8081

# Step 6: Display information
echo "=== Container Information ==="
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"

echo "=== Network Information ==="
echo "From VM2 (same VPC), access via:"
echo "  Apache: http://10.0.1.100:8080"
echo "  Nginx:  http://10.0.1.100:8081"

echo "From VM3 (different network), access via:"
echo "  Apache: http://203.0.113.10:8080"
echo "  Nginx:  http://203.0.113.10:8081"
```

Key Takeaways

1. **Docker Bridge Network (172.17.0.0/16)** is isolated to the host VM
2. **Port mapping (-p)** exposes container ports to host network
3. **VM2 (same VPC)** accesses via VM1's private IP (10.0.1.100)
4. **VM3 (different network)** accesses via VM1's public IP (203.0.113.10)
5. **Security groups/firewalls** must allow inbound traffic on mapped ports
6. **Direct container IPs** (172.17.0.x) are NOT accessible from other VMs

Understanding these networking concepts is crucial for deploying containerized applications in multi-VM, multi-VPC environments! 🚀