Activity 4: Networking in Docker (without Compose)

🎯 Objective

- Understand how Docker networking enables communication between containers and host.
- Compare **default bridge**, **user-defined bridge**, and special modes (`none`, `host`).
- Practice manual multi-container integration (**MySQL + phpMyAdmin**) without Compose.
- Apply **inspection tools** and **security best practices** for container networking.

✅ Prerequisites

- Completion of **Activity 3** (Building & Publishing Images).
- A running **Ubuntu 24.04 LTS EC2 instance** with Docker installed.
- SSH access to EC2 instance (`public-ip` and `secret-key.pem`).
- Internet access on EC2 instance to pull images.
- AWS Security Group (`docker-lab-sg`) updated to allow inbound traffic for:
    - **Port 8080** (phpMyAdmin).
    - Any additional ports you choose for testing.

📂 Lab Setup

- Your lab directory contains below important file:
    - `system-clean-init.sh` → helper script that prepares your EC2 environment for this activity.
- Required Docker images (to be pulled during tasks):
    - `nginx` (web server).

    - `alpine` (lightweight client container with ping/curl).
    - `mysql` (database service).
    - `phpmyadmin` (database admin UI).
- No source code needed for this activity → focus is purely on **networking**.

⚠️ **Important Note about `system-clean-init.sh`:** Running this script will **reset/clean** your EC2 instance environment. Use it carefully, since it wipes existing Docker containers/images and resets the workspace. For more understanding go through the well commented script.

📦 About the Setup

Instead of coding an application, this activity uses **ready-made service containers**:

- **nginx** → to simulate a web service.
- **alpine** → as a lightweight debugging client.
- **mysql** → relational database.
- **phpmyadmin** → web UI for MySQL.

By combining these containers manually, you will see how Docker networking allows:

- Containers to talk **within the same network**.
- The host machine (and external browser) to access containers via **port publishing**.
- DNS resolution and inspection of **network topology**.

⚠️ Without Compose, you will manually connect containers to networks, define environment variables, and publish ports. This helps you understand how **Compose** simplifies these steps later (Activity 5).

🧭 How we'll learn in this activity

We will progress step by step:

1. Explore the **default bridge network** (containers talk only via **IP**).
2. Create a **user-defined bridge network** (containers resolve each other by **name**).
3. Run **MySQL + phpMyAdmin** in the same network (manual integration + port publishing).
4. **Inspect networks** with Docker CLI (**subnets**, **DNS**, **connect/disconnect**).
5. Experiment with **special modes** (`none`, `host`) and review networking best practices.

By the end, you will have a solid grasp of **Docker's native networking model**, how to troubleshoot connectivity, and how to apply **secure networking practices** in real-world setups.

📝 *Notes*

- **Default vs User-defined bridge:**
  - Default bridge is limited (no DNS).
  - User-defined bridge adds DNS + easier service discovery.
- **Port Publishing:**
  - Publishing is only needed for host/external access.
  - Containers in the same network don't need ports exposed.

🐳 Task 1 — Explore the Default Bridge Network

🎯 Goal

- Understand how Docker's **default bridge network** works.
- Observe that containers in the default bridge can only talk to each other using **IP addresses**, not container names.
- Learn how to inspect container **networking** details using `docker inspect`.

📋 Explanation

- **Default Bridge Network** → When Docker is installed, it automatically creates a network named **`bridge`**.
    - All containers started **without specifying `--network`** are attached to this network.
    - Containers here can talk to each other, but **only via IP addresses** (no built-in DNS name resolution).
    - This is different from user-defined bridge networks, which we will explore in Task 2.

- **`docker inspect <container>`** → Used to fetch container details, including:
    - IP address.
    - Network name it is connected to.
    - Gateway and subnet details.

- **`ping` and `curl` inside containers** → Used to test connectivity.
    - `ping <ip>` → Tests basic ICMP reachability.
    - `curl http://<ip>` → Tests HTTP connectivity to nginx running inside a container.

🛠️ Steps

1. **Run an nginx container** (default bridge, port 80 exposed to container only, not host):

```
Shell
docker run -d --name web nginx
```

2. **Run an alpine container** (lightweight, attach to default bridge, interactive shell):

```
Shell
docker run -it --name client alpine sh
```

⚠️ Alpine does not have `curl` by default. Install it inside the container:

```
Shell
apk add --no-cache curl
```

3. **Find nginx container's IP address** (from another terminal, outside client):

```
Shell
docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' web
```

Suppose the IP is `172.17.0.2`.

4. **From alpine container, ping nginx**:

```
Shell
ping -c 3 172.17.0.2
```

✅ Expected: you should see ICMP replies.

5. **From alpine container, curl nginx**:

```
Shell
curl http://172.17.0.2
```

✅ Expected: HTML response from nginx's default welcome page.

6. **Try to curl using container name**:

```
Shell
ping -c 3 web
curl http://web
```

❌ Expected: This fails in default bridge (no DNS).

🖊️ Student Action (ans.json)

```JSON
{
  "lab4-defaultnetwork-type": "<bridge/host/none>",
  "lab4-defaultnetwork-nginx-ip": "<?>",
  "lab4-defaultnetwork-ping-success": "<true/false>",
  "lab4-defaultnetwork-curl-works-with-ip": "<true/false>",
  "lab4-defaultnetwork-curl-works-with-name": "<true/false>"
}
```

📌 Verification Checklist

- ✅ Nginx container (`web`) is running in background.
- ✅ Alpine container (`client`) is running and has `curl` installed.
- ✅ You can `ping` nginx container **by IP**.
- ✅ You can `curl` nginx container **by IP** and see HTML output.
- ❌ You cannot `curl` nginx container by **name** (`web`).

🐳 Task 2 — Create and Use a User-Defined Bridge Network

🎯 Goal

- Create a **user-defined bridge network** and understand why it is preferred over the default `bridge`.
- Attach containers to the custom network and verify **name-based DNS resolution** (container names resolve automatically).
- Inspect the network to observe DNS entries and container IP assignments.

📋 Explanation

- **`docker network create --driver bridge <name>`** → Creates a new user-defined bridge network. User-defined bridge networks provide built-in DNS, automatic name resolution, and better isolation/segmentation than the default `bridge`.
    - **Why use it:** containers attached to the same user-defined bridge can resolve each other by container **name** (e.g., `web`), enabling convenient service discovery without needing IP addresses.
    - **Driver**: `bridge` is the common driver for single-host networking. Other drivers (overlay, macvlan) exist for multi-host or advanced scenarios, but are out of scope here.
- **`--network <network>`** flag on `docker run` → Attaches the container to a specific network at creation time.
- **DNS resolution behavior:** Docker injects entries into the **embedded DNS server** for containers connected to a user-defined network. This is what enables `curl http://web` to succeed (from another container on the same network).
- **`docker network inspect <name>`** → Shows **subnet**, **gateway**, and `Containers` map with container names and IPs — useful to verify DNS entries and IP assignments.

🛠️ Steps

1. **Create a user-defined bridge network** called `lab4-net`:

```shell
docker network create --driver bridge lab4-net
```

2. **Run nginx** attached to `lab4-net` (detached, name `web`):

```
Shell
docker run -d --name web2 --network lab4-net nginx
```

3. **Run alpine** attached to the same network for testing (interactive shell):

```
Shell
docker run -it --name client2 --network lab4-net alpine sh
```

Inside the alpine shell, install `curl` (and optionally `iputils` for ping on some alpine variants):

```
Shell
apk add --no-cache curl iputils
```

4. **Verify nginx IP and network membership** from the host (outside the `client` container):

```
Shell
docker inspect -f '{{range $k,$v := .NetworkSettings.Networks}}{{$k}} ->
{{$v.IPAddress}}{{end}}' web2
# or
docker network inspect lab4-net
```

5. **From inside the alpine `client` shell, test name resolution and connectivity**:

```
Shell
# DNS by name (expected to succeed on user-defined bridge)
curl -sI http://web2 | head -n 5

# Ping by name (ICMP)
ping -c 3 web2
```

6. **Compare with curl by IP (optional):**

```shell
# find IP (host or via docker inspect) and curl:
curl -sI http://172.18.0.2 | head -n 5
```

7. **From the host, inspect the network mapping of containers**:

```shell
docker network inspect lab4-net --format='{{json .Containers}}' | jq
```

🖊️ Student Action (ans.json)

```json
{
  "lab4-userdefinednetwork-name": "<?>",
  "lab4-userdefinednetwork-driver": "<?>",
  "lab4-userdefinednetwork-web-ip": "<?>",
  "lab4-userdefinednetwork-curl-works-with-name": "<true/false>",
  "lab4-userdefinednetwork-ping-works-with-name": "<true/false>",
  "lab4-userdefinednetwork-subnet": "<?>",
  "lab4-userdefinednetwork-gateway": "<?>",
}
```

📌 Verification Checklist

- ✅ `lab4-net` network exists (`docker network ls` shows `lab4-net`).
- ✅ `web2` container is attached to `lab4-net` (`docker inspect web2` shows `lab4-net` under `NetworkSettings`).
- ✅ From `client2`, `curl http://web2` returns HTTP headers / success (HTTP 200 or default nginx headers).
- ✅ From `client2`, `ping web2` receives ICMP replies.
- ✅ `docker network inspect lab4-net` shows both `web2` and `client2` under `Containers` with assigned IP addresses.

- ✅ Student provided `lab4-userdefinednetwork-*` key-values in `ans.json`.

🐳 Task 3 — Multi-Container Integration (MySQL + phpMyAdmin)

🎯 Goal

- Run a **MySQL** server and **phpMyAdmin** as separate containers and connect them manually on the same user-defined network.
- Expose phpMyAdmin to the host so you can access the UI from a browser.
- Learn practical port-publishing patterns (`-p 8080:80` and dynamic mapping `-p :80`) and security considerations (AWS SG + minimal published ports).
- Capture runtime evidence (container IDs, IPs, host port) needed by the autograder.

📋 Explanation

- **MySQL container**
    - Use the official `mysql` image. Provide `MYSQL_ROOT_PASSWORD` (required) and optionally `MYSQL_DATABASE`, `MYSQL_USER`, `MYSQL_PASSWORD` to create a non-root DB user at first start.
    - MySQL initializes on first run; the container may take several seconds to become ready. You can check readiness with `docker logs` (look for "ready for connections") or `docker exec <mysql> mysqladmin ping -uroot -p"$MYSQL_ROOT_PASSWORD"`.
    - Persisting data with a volume is recommended in production but not required for this lab; the autograder will validate ephemeral containers.
- **phpMyAdmin container**

    - Use official `phpmyadmin` image. Configure connection target using `PMA_HOST=<mysql_container_name>` (or IP) and `PMA_USER`/`PMA_PASSWORD` if needed.
        - **`PMA_HOST`** → target MySQL container name (required).
        - **`PMA_USER` / `PMA_PASSWORD`** → optional defaults to auto-fill login (otherwise entered manually).
    - To access phpMyAdmin from your laptop/browser, publish the container port to the host with `-p <host_port>:80`. Example: `-p 8080:80`.
    - **Container-to-container** traffic (phpMyAdmin → MySQL) happens on the **user-defined bridge network** without any `-p`. Only **host→container** requires publishing.

- **Port publishing patterns**

    - `-p 8080:80` → deterministic host port `8080` maps to container `80`. Use this when you want a known URL (`http://<public-ip>:8080`).
    - `-p :80` (or `-p 0:80`) → Docker assigns a random host port; discoverable via `docker ps` or `docker port`. Useful when avoiding port conflicts; must query which host port was chosen.
    - **Security note:** Publish only the port(s) you need (in this task, only phpMyAdmin). Ensure the **EC2 security group** allows the chosen host port (e.g., 8080).

- **Networking**

    - Attach both containers to the same user-defined network (e.g., `lab4-net`) so phpMyAdmin can resolve MySQL by container name (e.g., `mysql-db`).
    - If MySQL and phpMyAdmin are on different networks, you must use `docker network connect` or publish ports—both are suboptimal for local service discovery.

- **Troubleshooting tips**

    - If phpMyAdmin shows “Error” or cannot connect:

        - Check `docker logs mysql-db` for initialization errors.

        - Verify `PMA_HOST` exactly matches the MySQL container name (**case-sensitive**).
        - Confirm MySQL readiness (look for “ready for connections”). If not ready, phpMyAdmin will fail until MySQL finishes init.

    - Use `docker exec -it mysql-db mysql -uroot -p"$MYSQL_ROOT_PASSWORD" -e "SELECT 1;"` to verify DB access from host.

    - Use `docker logs phpmyadmin` to inspect phpMyAdmin errors (e.g., unable to resolve host).

🛠️ Steps

Precondition: Ensure `lab4-net` exists (created in Task 2). If not, create it:

```
Shell
docker network create --driver bridge lab4-net
```

1. **Run MySQL container** (named `mysql-db`) on `lab4-net`:

```
Shell
docker run -d \
  --name mysql-db \
  --network lab4-net \
  -e MYSQL_ROOT_PASSWORD='rootpass123' \
  -e MYSQL_DATABASE='labdb' \
  -e MYSQL_USER='labuser' \
  -e MYSQL_PASSWORD='labpass123' \
  mysql:8.0
```

2. **Wait for MySQL to initialize** (you can poll readiness or watch logs):

```shell
# Option A: tail logs until ready (in separate terminal)
docker logs -f mysql-db

# Option B: poll mysqladmin (returns "mysqld is alive" when ready)
docker exec mysql-db mysqladmin ping -uroot -prootpass123 --silent
# repeat until exit code 0
```

3. **Run phpMyAdmin** attached to the same `lab4-net` and publish **host** port `8080`:

```shell
docker run -d \
  --name phpmyadmin \
  --network lab4-net \
  -e PMA_HOST='mysql-db' \
  -e PMA_USER='labuser' \
  -e PMA_PASSWORD='labpass123' \
  -p 8080:80 \
  phpmyadmin/phpmyadmin:latest
```

4. **Demonstrate dynamic/random port mapping** — run a second phpMyAdmin instance with Docker assigning a **host** port:

```shell
docker run -d \
  --name phpmyadmin-random \
  --network lab4-net \
  -e PMA_HOST='mysql-db' \
  -e PMA_USER='labuser' \
  -e PMA_PASSWORD='labpass123' \
  -p :80 \
  phpmyadmin/phpmyadmin:latest
```

Discover the assigned host port:

```shell
docker ps --format "table {{.Names}}\t{{.Ports}}"
```

```
# or
docker port phpmyadmin-random 80
```

5. **Verify phpMyAdmin from your browser** (on your laptop):
    - Visit: `http://<EC2-public-ip>:8080` → You should see phpMyAdmin page.
6. **Verify container-to-container connectivity (no -p needed)** from inside `phpmyadmin` container shell (optional):

```
Shell
# Install ping and curl inside phpMyAdmin (Debian-based)
docker exec -it phpmyadmin sh -c "apt-get update && apt-get install -y curl
iputils-ping netcat-openbsd >/dev/null 2>&1"

# 1. Ping MySQL by container name (tests DNS + ICMP reachability)
docker exec -it phpmyadmin ping -c 3 mysql-db

# 2. Check DNS resolution entry for mysql-db
docker exec -it phpmyadmin getent hosts mysql-db

# 3. Test MySQL port is open. Use netcat to check TCP connect.
docker exec -it phpmyadmin nc -vz mysql-db 3306
```

7. **Capture runtime evidence** (commands to run on host to fill `ans.json`):

```
Shell
# MySQL container IP
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
mysql-db

# phpMyAdmin container IP
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
phpmyadmin

# Which host port maps to phpMyAdmin (deterministic)
```

```
docker port phpmyadmin 80

# For dynamic mapping
docker port phpmyadmin-random 80

# Verify mysql readiness quickly
docker exec mysql-db mysqladmin ping -uroot -prootpass123
```

Do not remove containers — autograder will validate the running setup.

🖊️ Student Action (ans.json)

```JSON
{
  "lab4-mysql-container-name": "<?>",
  "lab4-mysql-ip": "<?>",
  "lab4-phpmyadmin-container-name": "<?>",
  "lab4-phpmyadmin-ip": "<?>",
  "lab4-phpmyadmin-host-port": "<?>",
  "lab4-phpmyadmin-random-host-port": "<?>",
  "lab4-phpmyadmin-accessible-from-host": "<true/false>",
  "lab4-container-to-container-connection-works": "<true/false>"
}
```

Hints for filling fields:

- `lab4-phpmyadmin-host-port` — result of `docker port phpmyadmin 80` (e.g., `0.0.0.0:8080`). Fill only the numeric host port (`8080`).

📌 Verification Checklist

- ✅ `mysql-db` container is running and attached to `lab4-net`.
- ✅ `phpmyadmin` container is running and attached to `lab4-net`.

- ✅ `docker logs mysql-db` shows initialization completed / “ready for connections”.
- ✅ `docker port phpmyadmin 80` shows `0.0.0.0:8080`.
- ✅ From your browser, `http://<EC2-public-ip>:8080` shows phpMyAdmin login page.
- ✅ Container-to-container resolution works: phpMyAdmin can resolve `mysql-db` (no host port needed).
- ✅ `ans.json` populated with the requested keys and correct values.

📝 *Notes:*

Port Reuse Across Networks

- Each Docker network provides an isolated namespace.
- This means **containers on different networks can reuse the same container port** (e.g., multiple MySQL containers, each on a different network, all listening on port `3306`).
- No conflict occurs internally because each network maintains its own **routing table** and IP address space.

- **How Different Networks Communicate**
  - By default, **containers on different user-defined networks cannot reach each other directly**, even if they expose the same port number.
  - To enable communication, you have two main options:

Connect one container to multiple networks

A container can be attached to more than one network using:

```shell
docker network connect <network-name> <container-name>
```

This gives the container **an IP in each network**, allowing it to talk to services in both.

```
None
frontend-app  <--->  api-server  <--->  backend-db
(frontend-net)        (dual-homed)       (backend-net)
```

Example sequence:

```
Shell
docker network create frontend-net
docker network create backend-net
docker run -d --name frontend-app --network frontend-net nginx
docker run -d --name backend-db --network backend-net mysql:8.0
docker run -d --name api-server --network frontend-net my-api-image
docker network connect backend-net api-server
docker inspect api-server
```

Now `api-server` can reach both `frontend-app` and `backend-db`, acting as a **bridge** between the two isolated networks.

Publish ports and access via the **host's IP**

- Example: publish MySQL with `-p 3307:3306` and access it from another container via `http://<host-ip>:3307`.
- This approach is less efficient (traffic goes through host) but works if direct network connection isn't possible.

Best Practice

- Keep related services (like MySQL and phpMyAdmin) inside the **same user-defined network**.
    - Use multi-network attachments only when a service must interact with multiple isolated environments (e.g., reverse proxy bridging frontend and backend).
- Avoid unnecessary port publishing — it expands the host's attack surface. Prefer **internal networking with container names** for service-to-service communication.

🐳 Task 4 — Inspect and Analyze Networks

🎯 Goal

- Use Docker CLI tools to inspect, analyze, and monitor networks.
- Understand what metadata Docker stores about networks (subnet, gateway, connected containers).
- Verify how DNS-based service discovery is working under the hood.

📋 Explanation

- **`docker network ls`** → Lists all networks on your host. You will see:
    - The **default bridge**, **host**, and **none** networks. (Will discuss more in Task5)
    - Any **user-defined networks** you created (e.g., `lab4-net`).
- **`docker network inspect <network>`** → Shows details about a network, including:
    - **Driver** (e.g., bridge).
    - **Subnet and Gateway** IPs.
    - **Containers** currently attached (with names, IDs, IPs).
    - Useful for verifying whether containers are correctly attached.
- **`docker inspect <container>`** → Can also be used at container level to confirm which networks it belongs to and what IPs it has. (Here search and explore field named: `NetworkSettings.Networks`)
- **DNS verification**
    - Containers in a user-defined bridge are registered with Docker's **embedded DNS server**.
    - This allows commands like `ping` **`mysql-db`** or `getent hosts phpmyadmin` inside a container to resolve names.

    - **`getent hosts <container>`** → shows how the container name resolves to an IP address. (must run from inside the connected container)

- **`docker events`** (optional advanced) → Streams live Docker events (including when containers connect/disconnect from networks). Helpful for debugging.

🛠️ Steps

1. **List all networks on your host:**

```
Shell
docker network ls
```

Expected: you should see at least these:

- `bridge` (default)
- `host`
- `none`
- `lab4-net` (from Task 2/3)

2. **Inspect the user-defined network (`lab4-net`):**

```
Shell
docker network inspect lab4-net | jq
```

Look at:

- `"Subnet"`
- `"Gateway"`
- `"Containers"` section (should list `mysql-db`, `phpmyadmin`, and `phpmyadmin-random`).

3. **Inspect MySQL container (`mysql-db`) networking info:**

```shell
docker inspect mysql-db --format='{{json .NetworkSettings.Networks}}' | jq
```

4. **Verify DNS resolution inside a container (`phpmyadmin`):**

```shell
docker exec -it phpmyadmin getent hosts mysql-db
docker exec -it phpmyadmin getent hosts phpmyadmin-random
```

Expected: each command returns IP + name mappings.

5. **(Optional) Monitor Docker events** while disconnecting/reconnecting containers:

```shell
docker events --filter type=network
```

Open another terminal, then:

```shell
docker network disconnect lab4-net phpmyadmin
docker network connect lab4-net phpmyadmin
```

Watch how `docker events` logs these changes.

🖊️ Student Action (ans.json)

```json
{
  "lab4-networkls-names": "<?>",
  "lab4-getent-mysql-db": "<?> (resolved IP from phpmyadmin)",
  "lab4-getent-phpmyadmin-random": "<?> (resolved IP from phpmyadmin)"
}
```

📌 Verification Checklist

- ✅ `docker network ls` shows all default + custom networks.
- ✅ `docker network inspect lab4-net` shows correct subnet, gateway, and containers.
- ✅ `docker inspect mysql-db` confirms it is attached to `lab4-net` with correct IP.
- ✅ `getent hosts mysql-db` from inside phpMyAdmin resolves to MySQL's IP.
- ✅ `getent hosts phpmyadmin-random` resolves to the random phpMyAdmin container's IP.
- ✅ All `ans.json` keys filled with correct values.

🐳 Task 5 — Special Network Modes (`none` and `host`) + Best Practices

🎯 Goal

- Learn the behavior and implications of Docker's special network modes: `--network none` and `--network host`.
- Observe how `host` mode gives the container direct access to the host network stack (and the risks that entails).
- Observe how `none` mode gives the container no network at all (strict isolation).

📋 Explanation

- **`--network none`**

    - Creates a container with **no network interfaces** (except `lo` in some kernels). The container cannot reach other containers or the outside world.
    - Use-cases: tight isolation for **compute-only** workloads, **security sandboxes**, or when you want a container that must not have network access.
    - Limitations: you cannot `ping`, `curl`, or `getent` out of a `none`-networked container unless you explicitly `docker network connect` later.

- `--network host` (Linux behavior — applies to EC2 Ubuntu hosts)

    - The container **shares the host's network namespace**. It sees the same network interfaces and IP addresses as the host.

    - Pros: **lowest latency**, no port mapping required (`-p` is ignored). Good for performance-sensitive networking workloads (raw sockets, network sniffers).

    - Cons / Security risks:

        - **No network isolation** — if the container is compromised, the attacker has direct access to host network interfaces.
        - **Port conflicts** — a service inside the container listening on port `80` will bind the host port `80` directly and may conflict with host services.
        - **Service discovery differences** — container cannot rely on Docker embedded DNS for name resolution the same way user-defined bridges do; name-based discovery between containers does not apply.

    - Note: On non-Linux platforms (Docker Desktop for Mac/Windows), `--network host` behaves differently or is limited — here we assume an Ubuntu EC2 host.

- **When to use which**

    - Use `none` for the strictest level of network isolation (e.g., untrusted workload, compute-only tasks).

- Use `host` only when you explicitly need host-level networking performance or must bind to host network interfaces; otherwise prefer user-defined bridge networks for isolation and service discovery.

🛠️ Steps

> **Warning:** In `--network host` mode, services bind directly to host ports. Be careful not to collide with essential host services (SSH, systemd services). Prefer test ports (e.g., 18080) or stop conflicting host services first on disposable lab VMs.

1. **Test `none` network — container should have no external network**

```shell
# Start a container with no network
docker run -d --name none-test --network none --rm alpine sleep 999999

# Exec into it and try network commands
docker exec -it none-test sh -c "ip addr || true"
docker exec -it none-test sh -c "apk add --no-cache iproute2 iputils || true"
docker exec -it none-test sh -c "ping -c 1 8.8.8.8 || true"
docker exec -it none-test sh -c "getent hosts google.com || true"

# Inspect container network settings (note: NetworkMode shows none)
docker inspect none-test --format '{{json .HostConfig.NetworkMode}}'
```

Expected:

- `ip addr` shows minimal interfaces (often only `lo`) or none.
- `apk add,ping` and `getent` fail (no network).

2. **Test `host` network — container shares host networking**

```shell
# Run a simple HTTP server in host mode binding to port 18080
```

```
docker run -d --name host-test --network host --rm alpine sh -c "apk add
--no-cache python3 && python3 -m http.server 18080 & sleep 999999"

# From host, confirm the server is reachable (no -p required)
curl -sS http://localhost:18080 || true

# Inspect container network settings (note: NetworkMode shows host)
docker inspect host-test --format '{{json .HostConfig.NetworkMode}}'
```

Expected:

- `curl http://localhost:18080` (run from host or another host process) returns HTTP response.
- `docker inspect` shows `HostConfig.NetworkMode` is `host`.
- `docker ps` will not show a published port for this container (because `-p` is ignored), but the service is available on the host IP/port.

3. **Demonstrate port conflict risk (do not run on SSH / critical ports!)**

```
Shell
# Attempt to run a container binding to host port 22 (SSH) — usually fails if
host SSH already bound
docker run -d --name dangerous --network host alpine sh -c "apk add --no-cache
python3 && python3 -m http.server 22 & sleep 99999" || true

#verify error with
docker logs dangerous
```

Expected:

- If host already has SSH bound on 22, your container will fail to bind and may exit with an error; this demonstrates risk of port conflicts. Don't run this on production hosts.

🖊️ Student Action (ans.json)

```JSON
{
  "lab4-none-container-name": "<?>",
  "lab4-none-ip-interfaces": "<?>",
  "lab4-none-can-ping": "<true/false>",
  "lab4-host-container-name": "host-test",
  "lab4-host-networkmode": "host",
  "lab4-host-service-port": "<?>",
  "lab4-host-accessible-from-host": "<true/false>",
  "lab4-host-port-conflict-observed": "<true/false>"
}
```

Hints:

- `lab4-none-ip-interfaces` — capture the list like `lo` / `none`.
- `lab4-host-accessible-from-host` — result of `curl http://localhost:18080` on the host (true/false).
- `lab4-host-port-conflict-observed` — true if any container failed to bind a host port because the host already had it in use.

📌 Verification Checklist

- ✅ `none-test` exists and shows no network connectivity (`ip addr` minimal, ping fails).
- ✅ `host-test` is running with `NetworkMode` set to `host`.
- ✅ Service started inside `host-test` is reachable from the host without `-p` (e.g., `curl http://localhost:18080`).
- ✅ `docker inspect` outputs contain the expected `NetworkMode` values for each container.
- ✅ `ans.json` populated with required keys.

🧠 Security & Best Practices (summary)

- Prefer **user-defined bridge** networks for application containers — they provide DNS-based service discovery and isolation.
- Use `--network none` for containers that must not have network access (untrusted tasks).
- Use `--network host` **only when necessary** (performance reasons or direct access to host interfaces). Understand that `host` mode removes network isolation and increases attack surface.
- Avoid running containers in `host` mode on production infrastructure unless you have a clear justification and compensating controls (firewalls, minimal privileges).
- Minimize port publishing (`-p`) to what is strictly required and keep your EC2 security group rules restrictive (only allow inbound on required host ports).

📌 Final Verification Checklist

✅ Containers on **default bridge** communicate only by IP (DNS fails).
✅ Containers on **user-defined bridge (`lab4-net`)** communicate by name (DNS works).
✅ `mysql-db`, `phpmyadmin`, and `phpmyadmin-random` run correctly on `lab4-net`.
✅ phpMyAdmin is accessible from browser at `http://<EC2-public-ip>:8080`.
✅ DNS resolution inside containers works using `getent hosts`.
✅ `none-test` container shows no network connectivity.
✅ `host-test` container shares host networking (service reachable at `localhost:18080`).
✅ `ans.json` contains all required keys:

- `lab4-defaultnetwork-type`
- `lab4-defaultnetwork-nginx-ip`
- `lab4-defaultnetwork-ping-success`
- `lab4-defaultnetwork-curl-works-with-ip`
- `lab4-defaultnetwork-curl-works-with-name`
- `lab4-userdefinednetwork-name`
- `lab4-userdefinednetwork-driver`
- `lab4-userdefinednetwork-web-ip`
- `lab4-userdefinednetwork-curl-works-with-name`

- `lab4-userdefinednetwork-ping-works-with-name`
- `lab4-userdefinednetwork-subnet`
- `lab4-userdefinednetwork-gateway`
- `lab4-mysql-container-name`
- `lab4-mysql-ip`
- `lab4-phpmyadmin-container-name`
- `lab4-phpmyadmin-ip`
- `lab4-phpmyadmin-host-port`
- `lab4-phpmyadmin-random-host-port`
- `lab4-phpmyadmin-accessible-from-host`
- `lab4-container-to-container-connection-works`
- `lab4-lab4net-connected-containers`
- `lab4-getent-mysql-db`
- `lab4-getent-phpmyadmin-random`
- `lab4-none-container-name`
- `lab4-none-ip-interfaces`
- `lab4-none-can-ping`
- `lab4-host-container-name`
- `lab4-host-networkmode`
- `lab4-host-service-port`
- `lab4-host-accessible-from-host`
- `lab4-host-port-conflict-observed`

🧹 Cleanup After the Activity

1. Stop and remove all containers/images:

```shell
docker ps -aq | xargs docker rm -f
docker images -aq | xargs docker rmi -f
```

2. Remove custom network:

```shell
docker network rm lab4-net
```

3. Remove/Prune all Caches (optional):

```shell
docker buildx prune -f
docker image prune -a -f
docker system prune -a --volumes -f
```

4. 🚨 Remove all files from EC2 instance if needed:

```shell
rm -rf ~/docker_lab/*
```

🛑 Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:

```shell
exit
```

2. Stop or terminate the EC2 instance from the AWS Management Console if no longer required.

🧠 Glossary (terms used above)

Term / Instruction	Meaning & Usage
`bridge` **Network**	Default network driver; isolates containers but allows manual IP connectivity.
User-defined Bridge	Custom bridge with built-in DNS, supports container name resolution.
Port Publishing (`-p`)	Maps a host port to a container port, enabling external access.
PMA_HOST / PMA_USER / PMA_PASSWORD	phpMyAdmin environment variables to configure MySQL connection target and default login.
`docker network ls`	Lists all networks available on the Docker host.
`docker network inspect`	Shows details of a network (subnet, gateway, connected containers).
`docker inspect <container>`	Displays container details, including network assignments.
DNS Resolution (`getent hosts`)	Verifies container name resolves to its IP within a user-defined network.
`--network none`	Starts container without any network access (except loopback).
`--network host`	Shares host's network namespace; no isolation, but high performance.
Port Conflict	When two services attempt to bind the same host port; leads to failure.
Embedded DNS	Internal Docker DNS server for resolving container names in user-defined networks.

Here's a **Note** section you can plug in after Task 5 (or at the very end before Glossary). It briefly mentions other Docker network drivers beyond `bridge`, `host`, and `none`:

📝 *Notes* — Other Docker Network Drivers

While this activity focused on **bridge**, **host**, and **none**, Docker provides several other drivers for advanced use-cases:

- **Overlay**
    - Connects containers across **multiple Docker hosts** using a distributed network.
    - Commonly used with **Docker Swarm** or orchestrators like **Kubernetes**.
    - Enables multi-host service discovery and communication without manual tunneling.
- **Macvlan**
    - Assigns containers their **own MAC address**, making them appear as physical devices on the local network.
    - Useful when you want containers to integrate directly into an existing physical LAN, each with its own IP.
    - Common in legacy systems that expect direct L2/L3 connectivity.
- **Ipvlan**
    - Similar to macvlan but more lightweight; shares the host's MAC address while assigning unique IPs at L3.
    - Useful in environments where MAC address limits exist (e.g., some cloud providers).
- **Overlay (Swarm/K8s)** and **Macvlan/Ipvlan** are less commonly needed in basic labs, but are crucial in **production-scale or enterprise networking scenarios**.

🏆 Congratulations — You’ve completed Activity 4!

END OF DOCUMENT