Activity 2: Docker Image Lifecycle & Persistence

🎯 Objective

In this activity, you will learn how to:

- Manage Docker images using **inspect**, **history**, **save**/**load**, and **commit**.
- Control container runtime behavior with **health checks**, **CPU/memory limits**, and **ulimits**.
- Use **volumes**, **bind** mounts, and **tmpfs** to persist data.
- Demonstrate the real-world usefulness of **persistence** with a MySQL database.

By the end, you will have hands-on experience managing images, applying runtime constraints, and configuring persistence in containers.

✅ Prerequisites

- Completion of **Activity 1** (basic container operations).
- A running **Ubuntu 24.04 LTS EC2 instance** with Docker installed.
- SSH access to EC2 instance using `public-ip` and `secret-key.pem` provided by you.
- Internet access on the EC2 instance to pull images.

🐳 Task 1: Image Management & Lifecycle

🎯 Goal

Work with Docker images to pull, inspect, analyze history, save/load images, and commit container changes.

📋 Do's

- Always verify image metadata with `inspect`.
- Use `history` to understand image layering.

- Demonstrate `save` and `load` workflow.
- Show how container state can be committed into a new image.

🛠️ Step-by-Step

1. Pull required images

```shell
docker pull mysql:latest
docker pull redis:latest
docker pull node:latest
```

What this does & why:

- `docker pull` downloads images from Docker Hub into the local repository.
- These images (MySQL, Redis, Node) represent real-world services commonly used in applications.
- Pulling ensures you can run containers from these images even without internet later.

🖊️ **Student Action:** Note down the **Image IDs** for `mysql:latest` and `redis:latest` into `ans.json`.

2. Inspect an image

```shell
docker image inspect node:latest
```

What this does & why:

- `docker image inspect` shows detailed metadata of an image in JSON format.
- Useful to view configuration like environment variables, entrypoints, exposed ports, architecture, and labels.

🖊️ **Student Action:** Extract the **Architecture** field value from the metadata of `redis:latest` image and record it in `ans.json`.

3. View history

```shell
docker history redis:latest
```

What this does & why:

- Displays the **layer history** of the image.
- Each line represents a layer created by a command in the Dockerfile (e.g., `RUN apt-get install`).
- Helps understand **image size contributors** and optimization opportunities.

🖊️ **Student Action:** Identify the **largest layer size** from the history output and note it in `ans.json`.

4. Save & load image

```shell
docker save -o /home/ubuntu/redis.tar redis:latest
docker rmi redis:latest
docker load -i /home/ubuntu/redis.tar
```

What this does & why:

- `docker save` exports an image as a `.tar` archive file.
- Useful for moving images between systems **without Docker Hub**.
- `docker rmi` removes the image locally.
- `docker load` restores the image from archive.

🖊️ **Student Action:**

- Record the **Node image ID** before `save` into `ans.json`.
- Ensure `node.tar` is created in `/home/ubuntu`.
- Confirm that the local repo no longer has `node:latest` after removal step.

5. Commit container state

```Shell
docker run -it --name test-commit ubuntu:latest bash
echo "persistent note" > /note.txt
exit
docker commit test-commit ubuntu-committed:lab2
```

What this does & why:

- `docker run -it` starts an interactive Ubuntu container.
- We create a new file (`/note.txt`) inside the container.
- `docker commit` creates a new Docker image from a modified or running container. This command captures the **current state of a container**, including any changes made to its **filesystem**, and saves it as a new image.
- This demonstrates how changes inside a running container can be preserved.

🖊️ **Student Action:** Record the **SHA256 digest** of the committed image (`ubuntu-committed:lab2`) in `ans.json`.

✅ Verification Checklist

- [ ] Pulled `mysql:latest`, `redis:latest`, `node:latest` and noted their Image IDs.
- [ ] Inspected `redis:latest` and recorded **Architecture**.
- [ ] Viewed history of `redis:latest` and noted **largest layer size**.
- [ ] Exported Node image → confirmed `node.tar` exists in `/home/ubuntu`.
- [ ] Removed Node image locally.
- [ ] Committed changes to Ubuntu container → recorded SHA256 digest.

```JSON
{
  "mysql-image-id": "sha256:<full_sha256>",
  "redis-image-id": "sha256:<full_sha256>",
  "redis-arch": "<x86_64/arm64/amd64>",
  "redis-largest-layer-size": "<x.yMB/x.yKB>",
```

```
  "node-image-id-tar": "sha256:<full_sha256>",
  "ubuntu-commit-sha256": "sha256:<full_sha256>"
}
```

📑 Useful Commands for this Task

Purpose	Command
Pull MySQL, Redis, Node images	`docker pull mysql:latest redis:latest node:latest`
Inspect image metadata	`docker image inspect <image>`
View image layer history	`docker history <image>`
Save image as tar	`docker save -o <file>.tar <image>`
Remove local image	`docker rmi <image>`
Load image from tar	`docker load -i <file>.tar`
Run container interactively	`docker run -it --name <name> <image> bash`
Commit container to new image	`docker commit <container> <new-image>`

🐳 Task 2: Container Behavior & Runtime Controls

🎯 Goal

Learn how to control runtime behavior of containers with **health checks**, **resource constraints**, and **file backups**.

📋 Do's

- Use `HEALTHCHECK` to monitor container state.
- Limit CPU and memory to simulate resource-constrained environments.

- Backup data from a container using `docker cp`.

🛠️ Step-by-Step

1. **Run Redis with a healthcheck**

```shell
docker run -d --name redis-health \
  --health-cmd="redis-cli ping || exit 1" \
  --health-interval=10s \
  --health-retries=3 \
  redis:latest

docker inspect --format='{{json .State.Health}}' redis-health
```

What this does & why:

- Adds a **healthcheck** to Redis so Docker can monitor its status.
- Docker periodically runs `redis-cli ping`. If it doesn't return `PONG`, the container is marked **unhealthy**.
- This is essential for automated recovery in real-world systems (e.g., restart unhealthy containers).

🖊️ **Student Action:** Record the **health status** (`healthy` or `unhealthy`) of `redis-health` in `ans.json`.

2. **Apply resource limits**

```shell
docker run -d --name limited-redis --cpus="0.5" --memory="256m" redis:latest

docker inspect limited-redis
```

What this does & why:

- Restricts the container to **0.5 CPU** and **256 MB memory**.
- Prevents a single container from consuming all system resources.

- This is useful in multi-container workloads or shared environments.

Other resources that can be limited with Docker:

- `--pids-limit` → restrict number of processes.
- `--blkio-weight` → control disk I/O priority.
- `--device-read-bps` / `--device-write-bps` → limit block device read/write rates.

🖊️ **Student Action:** From the container inspect output, find and record the following for `limited-redis`: `Memory` value; `NanoCpus` value. Add them to `ans.json`.

3. **Apply ulimit restriction**

```Shell
docker run -d --name limited-ulimit --ulimit nofile=1000:1000 redis:latest

docker inspect limited-ulimit
```

What this does & why:

- `--ulimit` sets **kernel resource limits** for containers.
- Here `nofile=1000:1000` restricts max open files (soft=1000, hard=1000) per process.
- **Soft limit** = the **current limit** the process is allowed to reach.
- **Hard limit** = the **maximum ceiling** that the soft limit can be raised to (only by root or privileged processes).
- Ulimits prevent resource exhaustion by rogue applications.

Meaning of `--ulimit`:

- It controls system resource limits for processes inside a container.
- Flags supported include:
  - `nofile` → max open files.
  - `nproc` → max number of processes.
  - `fsize` → max file size.
  - `cpu` → max CPU time.
  - `as` → max virtual memory.

        ◦ `core` → max core dump size.

🖊️ **Student Action:** Record the **nofile ulimit value** from container inspect in `ans.json`.

🌟 Extra Note: Resource Limits vs Ulimits

- **Resource Limits (`--cpus`, `--memory`, etc.)**
  - These control how much **system resources** (CPU, RAM, I/O) a container can consume.
  - Implemented via Linux **cgroups (control groups)**.
  - Ensures fair distribution of resources across containers.
  - Example: `--memory=256m` prevents a container from using more than 256MB RAM.

- **Ulimits (`--ulimit nofile=1000:1000`, etc.)**
  - These restrict **per-process limits** inside the container.
  - Implemented via **kernel-level limits** that apply to processes.
  - Prevents applications inside a container from exhausting OS-level resources.
  - Example: `--ulimit nofile=1000:1000` restricts open file descriptors per process.

Key Difference:

- *Resource limits* → control **how much** system resources a container can take.
- *Ulimits* → control **how processes behave** inside a container.

Together, they form a **two-layer defense**:

1. **Container-level control** (don't starve the host).
2. **Process-level safety** (don't let one app inside the container break everything).

Comparison Table

Feature	`--cpus`, `--memory`, `--pids-limit`	`--ulimit`
Control Mechanism	Linux Cgroups (Control Groups)	Linux Ulimits (User Limits)

Resource Type	CPU, Memory, PIDs	Process-level resources (CPU time, open files, process count, file size, etc.)
Scope of Limit	Applies to the entire container	Applies per-process within the container
Example for CPU	`--cpus=1.5` (limits CPU core usage)	`--ulimit cpu=10` (limits process runtime in seconds)
Primary Use Case	Managing system-wide resource allocation for containers to ensure fair usage and prevent host degradation.	Controlling individual process behavior to prevent resource leaks or denial-of-service from a misbehaving application.

4. **Backup file from container**

```shell
docker exec limited-redis sh -c 'echo "backup data" > /data.txt'
docker cp limited-redis:/data.txt ./copied_data.txt
```

What this does & why:

- `docker exec` creates a test file `/data.txt` inside the running container.
- `docker cp` copies that file from container to host.
- This is useful for **backups, debugging, or exporting logs/data** from containers without setting up volumes.

🖊️ **Student Action:** Verify the backed up file is present on the host at `/home/ubuntu`. Record the name of the file present on the host in `ans.json`.

✅ Verification Checklist

- ☐ Redis container (`redis-health`) reports health status (healthy/unhealthy).
- ☐ `limited-redis` container created with CPU and memory restrictions.
- ☐ `limited-ulimit` container runs with `nofile=1000` ulimit.
- ☐ File `data.txt` copied successfully to host.

```json
{
  "redis-health-status": "<value>",
  "limited-redis-nanocpus": "<value>",
  "limited-redis-memory": "<value>",
  "limited-ulimit-nofile": "soft={<value>},hard={<value>}",
  "backup-host-file-name": "<filename.ext>",
}
```

🐳 Task 3: Volumes & Persistence (MySQL Use Case)

🎯 Goal

Understand that containers are **ephemeral** by default (data is lost after container removal).
Learn how to persist data using **volumes**, and compare with **tmpfs** and **bind mounts**.

📋 Do’s

- Show that data does **not** persist without volumes.
- Use a **named volume** for MySQL data.
- Insert and query data from MySQL.
- Prove persistence after container recreation.
- Compare with **tmpfs** and **bind mounts**.

🛠️ Step-by-Step

1. **Start MySQL without a volume (non-persistent run)**

Shell

```shell
docker run -d --name mysql-temp \
  -e MYSQL_ROOT_PASSWORD=rootpass \
  -e MYSQL_DATABASE=school \
  mysql:latest
```

Shell

```shell
docker exec -it mysql-temp \
  mysql -uroot -prootpass \
  -e "USE school; CREATE TABLE students (id INT, name VARCHAR(50));
INSERT INTO students VALUES (1, 'Eve');"
```

Shell

```shell
docker exec -it mysql-temp \
  mysql -uroot -prootpass \
  -e "SELECT * FROM school.students;"
```

Shell

```shell
docker rm -f mysql-temp
```

Then start a **fresh MySQL container** (again without a volume):

Shell

```shell
docker run -d --name mysql-temp \
  -e MYSQL_ROOT_PASSWORD=rootpass \
  -e MYSQL_DATABASE=school \
  mysql:latest
```

```Shell
docker exec -it mysql-temp \
  mysql -uroot -prootpass \
  -e "SHOW TABLES IN school;"
```

What this shows & why:

- After removing the container, the data is **gone**.
- Confirms that container filesystems are ephemeral.

🖊️ **Student Action:** After recreating the container without volume, run `SHOW TABLES IN school;` Record whether the `students` table was found (`exists` / `notfound`) in `ans.json`.

2. **Create a named volume**

```Shell
docker volume create mysql_data
```

What this does & why:

- Creates a persistent Docker-managed storage location.
- Find the volume at host: `sudo ls /var/lib/docker/volumes/`
- Volumes survive container removal.

🖊️ **Student Action:** Record the **volume name** in `ans.json`.

3. **Run MySQL with the named volume**

```Shell
docker run -d --name mysql-persist \
  -e MYSQL_ROOT_PASSWORD=rootpass \
  -e MYSQL_DATABASE=school \
  -v mysql_data:/var/lib/mysql \
  mysql:latest
```

What this does & why:

- Runs MySQL with a named volume `mysql_data` attached at container's `/var/lib/mysql`.
- Ensures all database files are stored persistently.

🖊️ **Student Action:** Run `docker inspect mysql-persist` and record both: `Mounts.Source` (host **full** path of the volume); `Mounts.Destination` (container **full** path of the volume) in `ans.json`.

4. **Insert data into MySQL**

Shell

```
docker exec -it mysql-persist \
  mysql -uroot -prootpass \
  -e "USE school; CREATE TABLE students (id INT, name VARCHAR(50));
INSERT INTO students VALUES (1, 'Alice'), (2, 'Bob');"
```

5. **Query the data**

Shell

```
docker exec -it mysql-persist \
  mysql -uroot -prootpass \
  -e "SELECT * FROM school.students;"
```

🖊️ **Student Action:** Run `SELECT COUNT(*) FROM school.students;` and record the **row count** in `ans.json`.

6. **Test persistence with the volume**

Shell

```
docker rm -f mysql-persist # Remove container
```

```
docker run -d --name mysql-persist \    # Re-run container
  -e MYSQL_ROOT_PASSWORD=rootpass \
  -e MYSQL_DATABASE=school \
  -v mysql_data:/var/lib/mysql \
  mysql:latest

docker exec -it mysql-persist \  # Check if data persists
  mysql -uroot -prootpass \
  -e "SELECT * FROM school.students;"
```

What this does & why:

- Container removed, but volume kept.
- On restart, the data is still there → proves persistence.

🖊️ **Student Action:** After recreating the container with the same named volume, run the same row count query. Record whether the **row count before and after** is the same (`same` / `different`) in `ans.json`.

7. **Compare with tmpfs and bind mount**

   Tmpfs Mount (memory-only):

```shell
docker run -d --name mysql-tmpfs \
  -e MYSQL_ROOT_PASSWORD=rootpass \
  -e MYSQL_DATABASE=school \
  --mount type=tmpfs,target=/var/lib/mysql \
  mysql:latest
```

What this does & why:

- Stores MySQL data in **RAM** only (no disk write).
- Very fast, good for **caching or temporary workloads**.
- But all data is lost once the container stops/restarts.

- Not suitable for databases that require durability.

🖊️ **Student Action:** Insert rows into `mysql-tmpfs`, restart the container, then re-run the row count query. Record whether the **row count after restart** is `0` or `>0` in `ans.json`.

Bind Mount (host filesystem):

```Shell
docker run -d --name mysql-bind \
  -e MYSQL_ROOT_PASSWORD=rootpass \
  -e MYSQL_DATABASE=school \
  -v /home/ubuntu/mysql_data:/var/lib/mysql \
  mysql:latest
```

What this does & why:

- Maps a **host directory** into the container.
- Data is visible directly on the host filesystem (`/home/ubuntu/mysql_data`).
- Useful when you need to inspect/edit files directly, or for development/debugging.
- But less portable → depends on host directory structure and permissions.

🌟 **Extra Note: Why Volumes are better (default choice for DBs):**

- Managed by Docker, independent of host filesystem structure.
- More portable across environments.
- Backed up/restored easily with `docker volume` commands.
- Recommended for **production databases** like MySQL/Postgres.

🖊️ **Student Action:**

- Record the **bind mount path** `Mounts.Source`; `Mounts.Destination` in `ans.json`.

✅ Verification Checklist

- ☐ Recorded **no-volume-case** indicating table existence status after recreation.
- ☐ Recorded **mysql-volume-name** for the created named volume.
- ☐ Recorded **mysql-named-volume-mount-source** and **mount-destination**.
- ☐ Recorded **mysql-row-count** before recreation.
- ☐ Recorded **mysql-row-count-persistence-check** after recreation..
- ☐ Recorded **mysql-tmpfs-row-count-after-restart** recorded after restarting.
- ☐ Recorded **mysql-bind-mount-source** and **mount-destination**.

JSON
```json
{
  "no-volume-case": "exists/notfound",
  "mysql-volume-name": "",
  "mysql-named-volume-mount-source": "",
  "mysql-named-volume-mount-destination": "",
  "mysql-row-count": "",
  "mysql-row-count-persistence-check": "same/different",
  "mysql-tmpfs-row-count": "0/>0",
  "mysql-bind-mount-source": "",
  "mysql-bind-mount-destination": ""
}
```

📝 Final Verification Checklist

Task 1 — Basic Containers

- ☐ Pulled `mysql:latest`, `redis:latest`, `node:latest` and noted their Image IDs.
- ☐ Inspected `redis:latest` and recorded **Architecture**.
- ☐ Viewed history of `redis:latest` and noted **largest layer size**.
- ☐ Exported Node image → confirmed `node.tar` exists in `/home/ubuntu`.
- ☐ Removed Node image locally.
- ☐ Committed changes to Ubuntu container → recorded SHA256 digest.

Task 2 — CLI & Inspection

- ☐ Redis container (`redis-health`) reports health status (healthy/unhealthy).
- ☐ `limited-redis` container created with CPU and memory restrictions.
- ☐ `limited-ulimit` container runs with `nofile=1000` ulimit.
- ☐ File `data.txt` copied successfully to host.

Task 3 — Non-Root Containers

- ☐ Recorded **no-volume-case** indicating table existence status after recreation.
- ☐ Recorded **mysql-volume-name** for the created named volume.
- ☐ Recorded **mysql-named-volume-mount-source** and **mount-destination**.
- ☐ Recorded **mysql-row-count** before recreation.
- ☐ Recorded **mysql-row-count-persistence-check** after recreation..
- ☐ Recorded **mysql-tmpfs-row-count-after-restart** recorded after restarting.
- ☐ Recorded **mysql-bind-mount-source** and **mount-destination**.

🧹 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/Prune all Caches (optional):

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

3. 🚨 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.

🏆 Congratulations — You've completed Activity 2!

═════════════ END OF DOCUMENT ═════════════