Handout: Persistent Storage

Docker containers are ephemeral by design — changes made inside a container are lost when it stops or is removed. To persist data (like database files, user uploads, or logs), Docker offers **volumes** and **bind mounts**.

1. Volumes (Managed by Docker)

Volumes are the recommended mechanism for persistent data. To start a container and mount a volume:

```
docker run --mount type=volume,source=<name>,target=<container-path> <image>
```

If the volume doesn't already exist, Docker creates it automatically. For example, run a PostgreSQL container with a volume named `pgdata` for data storage like:

```
docker run -d \
  --name postgres-container \
  --mount type=volume,source=pgdata,target=/var/lib/postgresql/data \
  -e POSTGRES_PASSWORD=mysecretpassword \
  postgres
```

Volume data is stored in Docker's internal volume directory (e.g. `/var/lib/docker/volumes/`). However, on **Windows** and **macOS**, Docker runs inside a virtual machine. This means volume data is stored **inside the VM**, not directly on your host system — so you won't find it at that path from your regular file browser. Because volumes are fully managed by Docker and decoupled from the host filesystem, they are portable, easy to back up, and behave consistently across different environments.

Volumes must be attached when the container is created. You cannot add or remove volumes from an existing container. To change volume mounts, you need to stop and remove the container, then start a new one with the updated configuration.

There is also a shorter notation:

```
docker run -v <name>:<container-path> <image>
```

While commonly used, this shorthand is less explicit and not recommended by the official documentation.

1.1. Unnamed Volumes

Anonymous volumes are created when you use the `VOLUME` instruction in a Dockerfile:

```
VOLUME /data
```

This does not create a volume during image build — it simply tells Docker that when a container is started, the specified path should be backed by a volume. If no volume is explicitly provided at runtime, Docker will create a randomly named volume for you.

To keep control over your data and avoid orphaned volumes, it's best to manually map a named volume to the declared path.

1.2. Managing Volumes

You can manually create a named volume using:

```
docker volume create <name>
```

This allows you to define the volume explicitly and reuse it across multiple containers. To delete a volume:

```
docker volume rm <name>
```

To clean up dangling (unused) volumes:

```
docker volume prune
```

This removes all volumes that are no longer attached to any container.

A volume can only be removed if no container is using it. You must stop and remove all dependent containers before deleting it.

2. Bind Mounts (Host-Linked)

Bind mounts allow you to map a specific file or directory from your host machine directly into a container's filesystem:

```
docker run -v <host-path>:<container-path> <image>
```

For example, to mount a local data directory into `/app/data` inside the container:

```
docker run -v ${PWD}/data:/app/data myapp
```

In this setup, changes on the host are immediately reflected in the container, and vice versa.
Unlike volumes, bind mounts reference exact host paths and are not managed by Docker. Bind mounts can break across systems or teams due to differing host paths and OS-level permissions. Avoid them in production unless absolutely necessary.
While relative paths do work, you should always use absolute paths for bind mounts. Using `$(pwd)` (or `${PWD}` in some shells) ensures the path resolves to your current working directory, making behavior predictable and scripts portable.

Bind mounts, like volumes, must be attached at container creation. To change them, you must remove and recreate the container with the new configuration.

3. Comparison: Volumes vs. Bind Mounts

Feature	Bind Mounts	Volumes
Management	Tied to host filesystem, not managed by Docker	Fully managed by Docker, independent of host
Persistence	Depends on the host	Persistent; survives container removal
Creation	Requires explicit host path	Created with `docker volume create` or via `docker run`
Portability	Limited; relies on host-specific paths	Works identically across Docker-enabled systems
Use Case	Best for development (live code changes)	Ideal for production (clean, consistent, portable)
Performance	Can be slower or inconsistent (depends on host setup)	Generally faster and more predictable

Use **volumes** for production and **bind mounts** for development.