AI-03965 Docker Volumes Paths

Owner	Ⓡ Rigel Arcayan
Tags	Project
Created time	June 23, 2026 7:55 AM

Explain Docker Volumes path.

What does
./
/

When dealing with Docker, paths can get a bit confusing because you are constantly juggling two completely different environments: your **host machine** (your actual computer) and the **Docker container** (the isolated environment running your app).

Here is the breakdown of what `./` and `/` mean in the context of Docker volumes.

1. The Short Answer: `./` vs `/`

In a Docker volume or bind mount flag (`-v` or `--volume`), the syntax always follows the pattern: `[Host Path]:[Container Path]`.

- `./` **(Relative Path):** Refers to the **current working directory** on your host machine (where you are currently running the terminal command).
- `/` **(Absolute Path):** Refers to the **root directory** of the filesystem. Depending on which side of the `:` it is on, it's either the root of your host machine or the root of the container.

2. How They Behave in Docker Commands

Let's look at concrete examples to see exactly how Docker interprets these.

Using `./` (Relative Host Path)

> **Note:** Standard Docker CLI `-v` syntax historically required absolute paths, but modern Docker (and Docker Compose) supports relative paths using `./`.

Bash

```
docker run -v ./app:/usr/src/app my-image
```

- `./app`: Looks inside the *current folder* on your host machine for a directory named `app`.
- `/usr/src/app`: Points to an *absolute path* inside the container where that host folder will be mounted.

Using `/` (Absolute Paths)

Bash

```
docker run -v /var/log/app:/var/log/my-app my-image
```

- `/var/log/app`: Points to an exact, specific location starting from the root of your **host machine**.
- `/var/log/my-app`: Points to an exact, specific location starting from the root of the **container**.

3. Docker Compose Context (The most common usage)

In a `docker-compose.yml` file, `./` is incredibly common because it ensures your project is portable.

YAML

```
version: '3.8'
services:
  web:
    image: node:18
    volumes:
      # `./` means "look in the same folder as this docker-compose.yml file"
```

```
      - ./src:/app/src

      # `/` on the right side means an absolute path inside the container
      - /app/node_modules
```

Summary Cheat Sheet

Symbol	Context	Meaning	Example
`./`	Host Side (Left)	The directory you are currently standing in.	`./data` $\rightarrow$ folder in current directory
`/`	Host Side (Left)	A fixed, specific folder on your actual computer.	`/Users/username/data`
`/`	Container Side (Right)	A fixed, specific folder inside the isolated container.	`/var/www/html`

Rule of thumb: Always use `./` when you want to mount files from your project folder so that the command works for anyone else who downloads your code. Use `/` when you are targeting system-level directories.

So for a postgres or mariadb, what is the bkm for the path to my database? How would I keep my database safe and portable? I'm in GCP and I don't really know if I'm staying in GCP or not?

When managing a database like PostgreSQL or MariaDB inside Docker, the best known method (**BKM**) relies on two core principles: **separation of concerns** (decoupling the data from the container life cycle) and **infrastructure agnosticism** (making sure your configuration works whether you are on GCP, AWS, or a bare-metal server).

Here is how you handle paths, safety, and portability.

1. The BKM for Paths: Named Volumes vs. Bind Mounts

There are two primary ways to persist data in Docker, and each has a specific use case:

Method A: Docker Named Volumes (Recommended for Production & Cloud)

Instead of pointing to a specific path on your host machine, you let Docker manage the storage engine.

YAML

```
version: '3.8'

services:
  db:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: secretpassword
    volumes:
      # Format: [Named Volume]:[Container Internal Data Path]
      - pg_data:/var/lib/postgresql/data

volumes:
  pg_data: # Docker creates and manages this volume automatically
```

- **Postgres Container Path:** `/var/lib/postgresql/data`
- **MariaDB Container Path:** `/var/lib/mysql`

Why this is the BKM: Docker optimizes the filesystem performance for named volumes. You don't have to worry about host-level file permissions (which can get messy between Linux hosts and cloud providers), and Docker handles the underlying storage.

Method B: Relative Bind Mounts (Recommended for Local Dev)

If you must see the files on your host, use a relative path (`./`).

YAML

```
    volumes:
      - ./db_data:/var/lib/postgresql/data
```

- **Why use it:** Great for local development because deleting the folder resets the DB completely.
- **Why avoid in production:** It can cause permission issues across different OS hosts and cloud environments.

2. Keeping Your Data Safe and Portable

Since you are in GCP but want the flexibility to lift and shift to another cloud provider or an on-prem server tomorrow, your safety and portability checklist should look like this:

Use Docker Compose (Your Portability Blueprint)

Never run raw `docker run` commands in production. Keep everything in a `docker-compose.yml` file. Because Docker Compose abstracts the infrastructure, **your exact same YAML file will run perfectly on GCP Compute Engine, AWS EC2, DigitalOcean, or a home server.**

Decouple the Compute from the Storage

If you run Docker directly on a cloud virtual machine (like a GCP Compute Engine instance), do not store the database volume on the boot disk.

1. Attach a separate **Cloud Persistent Disk** to your VM.
2. Mount that disk to a specific directory on your host (e.g., `/mnt/disks/db-storage`).
3. Configure Docker to store its volumes there.

If you decide to leave GCP, you simply take a final snapshot or backup of the data, spin up a VM on your new provider, install Docker, copy the data over, and run `docker compose up -d`.

The Ultimate Safety Net: Logical Backups

While backing up raw filesystem volumes is fast, it isn't always perfectly portable between different database minor versions or operating systems. The gold standard for database safety is a regular automated cron job that spits out a logical backup script:

- **PostgreSQL:** `pg_dump` or `pg_dumpall`

- **MariaDB:** `mariadb-dump` (or `mysqldump`)

Store these `.sql` or `.tar` backup files in an object storage bucket (like GCP Cloud Storage). If you change cloud providers, object storage is incredibly easy to migrate, and a standard SQL dump can be restored onto *any* Postgres or MariaDB instance anywhere.

Summary Strategy for Cloud Flexibility

Objective	Action
Path BKM	Use a **Named Volume** (e.g., `pg_data:/var/lib/postgresql/data`).
Portability	Keep all configurations in a `docker-compose.yml` file. Avoid cloud-specific wrapper tools where possible.
Safety	Automate daily `pg_dump` / `mariadb-dump` files and send them to an external storage location.

How are you currently running Docker in GCP? If you're using standard virtual machines, we can look at setting up the automated backup script to make moving off GCP a breeze if you ever choose to.