1) Explore Docker Commands for Content Management

Goal: Learn how to **pull, inspect, copy, snapshot, save/load, export/import** content; and how to use **volumes** and **bind mounts** for persistence.

We'll use a tiny Linux image: **alpine**.

A. Image Management

1. **Pull an image**

```
docker pull alpine:3.19
```

2. **List images**

```
docker images
```

3. **Inspect image metadata** (dig into labels, layers, env, cmd)

```
docker inspect alpine:3.19
```

4. **View layer history**

```
docker history alpine:3.19
```

5. **Tag an image** (create a friendly alias)

```
docker tag alpine:3.19 alpine:lab
```

6. **Remove an image** (only when no container uses it)

```
docker rmi alpine:lab
```

7. **Save/Load images as .tar files** (portable archives)

```
# Save to tar
mkdir -p images && docker save -o images/alpine_3_19.tar
alpine:3.19
# Load back from tar
docker load -i images/alpine_3_19.tar
```

When to use: *save/load* keeps **image** + **metadata** (tags, history).

B. Container Content Management

1. **Create & start an interactive container**

```
# Start a shell in alpine, name it 'labc'
docker run -it --name labc alpine:3.19 sh
```

Inside the container:

```
# create files and directories
echo "hello from container" > /msg.txt
```

```
mkdir -p /data && echo "42" > /data/number.txt
cat /msg.txt; ls -R /
exit
```

2. **List containers**

```
docker ps -a
```

3. **Copy files to/from a container**

```
# From container to host
mkdir -p out && docker cp labc:/msg.txt ./out/msg.txt
# From host to container
echo "added-from-host" > host.txt
docker cp host.txt labc:/data/host.txt
```

4. **See what changed inside the container**

```
docker diff labc
```

5. **Inspect container metadata**

```
docker inspect labc
```

6. **View container logs** (useful for apps)

```
docker logs labc
```

7. **Execute a command in a running container**

```
docker exec -it labc sh -lc 'ls -l /data && cat /msg.txt'
```

8. **Snapshot container as a new image (commit)**

```
docker commit labc alpine:with-data
# verify new image exists
docker images | grep with-data
```

When to use: *commit* captures the container's **current filesystem**, useful for quick snapshots (but prefer **Dockerfile** for reproducible builds).

9. **Export/Import** (filesystem only, no image metadata/history)

```
# Export container's rootfs
docker export labc -o container_rootfs.tar
# Import as a new image
docker import container_rootfs.tar alpine:imported
```

When to use: *export/import* is a flat snapshot of the **container filesystem** (no layers, no history).

10. **Stop & remove containers**

```
docker stop labc && docker rm labc
```

C. Persistence with Volumes & Bind Mounts

1. **Named Volume** (managed by Docker)

```
# Create a volume and attach it at /appdata
```

```
docker volume create labvol
# Start a container using the volume
docker run -it --name volc -v labvol:/appdata alpine:3.19 sh
```

Inside the container:

```
echo "persist me" > /appdata/file.txt
exit
```

Now remove the container and reattach the same volume:

```
docker rm volc
# Start a new container with the *same* volume
docker run --rm -it -v labvol:/appdata alpine:3.19 sh -lc
'cat /appdata/file.txt'
# List volumes and inspect
docker volume ls
docker volume inspect labvol
```

2. **Bind Mount** (host directory mapped into container)

- **Linux/macOS**

```
mkdir -p $(pwd)/bind && echo "host content" > $(pwd)/bind/
host.txt
docker run --rm -it -v $(pwd)/bind:/mnt alpine:3.19 sh -lc
'ls -l /mnt && cat /mnt/host.txt'
```

- **Windows PowerShell**

```
mkdir bind | Out-Null; Set-Content -Path .\bind\host.txt
-Value 'host content'
docker run --rm -it -v ${PWD}\\bind:/mnt alpine:3.19 sh -lc
"ls -l /mnt && cat /mnt/host.txt"
```

Tip: Bind mounts are perfect for **live editing** your app code from the host.

3. **Cleanup**

```
docker volume rm labvol
# Review disk usage & prune
docker system df
docker system prune -f
```

1️⃣ Docker Use Cases Clarification

Docker is usually used for:

- Running **applications and services** in isolated environments.
- Ensuring **consistent environments** across development, testing, and production.
- Packaging **backend APIs, databases, microservices, or full stacks**.

Serving **just a static HTML file** is *possible* but in real-world projects you usually serve HTML **as part of a web app** (via Nginx, Apache, or a web framework).

So yes, you **can** do it with Docker, but Docker shines more when you're packaging apps that have **logic or dependencies**, not just a single HTML page.

2️⃣ Example: Containerized `event_registration.html`

Project structure

```
event-app/
├── Dockerfile
└── event_registration.html
```

`event_registration.html` (example)

```
<!DOCTYPE html>
<html>
<head>
  <title>Event Registration</title>
</head>
<body>
  <h1>Register for the Event</h1>
  <form>
    Name: <input type="text" name="name"><br><br>
    Email: <input type="email" name="email"><br><br>
    <button type="submit">Register</button>
  </form>
</body>
</html>
```

Dockerfile

We'll use **Nginx** to serve the static HTML:

```
# Use lightweight nginx image
FROM nginx:alpine

# Copy HTML into nginx default folder
COPY event_registration.html /usr/share/nginx/html/index.html

# Expose port 80
EXPOSE 80

# Start nginx (already default in image)
CMD ["nginx", "-g", "daemon off;"]
```

3️⃣ Build the Docker image

```
docker build -t event-app:1.0 .
```

4️⃣ Run the container

```
docker run -d -p 8000:80 --name event-container event-app:1.0
```

- `-p 8000:80` → host port 8000 maps to container port 80

5️⃣ Test it

Open in browser:
👉 http://localhost:8000

You should see your **event registration page**.

✅ This is exactly what you asked: the container serves a **static HTML file**.

Key Notes

- Docker **can** serve single HTML files, but it's more commonly used to serve **applications, microservices, or backend APIs**.
- For **just static HTML**, Docker adds isolation but might be overkill compared to just opening the file in a browser.
- Docker **shines** when your HTML is part of a larger app with dependencies, like Flask, Node.js, Django, etc.

3. Explanation of this line from the Dockerfile:

`CMD ["nginx", "-g", "daemon off;"]`

and you're asking:

"What is `daemon off;`?"

Let's break it down simply 👇

🧩 The context

Normally, **Nginx** (when run on a regular server) runs as a **daemon**, meaning it:

- Starts in the background
- Keeps running even after the shell closes

That's great for servers — but **not** for Docker.

🐳 Why "daemon off;" is used in Docker

In Docker, every container:

- Runs **a single main process** (called PID 1)
- Stops when that process ends

If Nginx starts as a background daemon, the container will think:

"The main process has exited — I should stop."

So the container shuts down immediately. ❌

✅ Solution:

We tell Nginx **not** to run as a background daemon.

That's what this part does:

```
nginx -g "daemon off;"
```

It overrides Nginx's default behavior and keeps it running **in the foreground** — so Docker sees it as an active process and keeps the container alive.

🔍 Breaking down the syntax

Part	Meaning
`nginx`	The Nginx command to start the web server
`-g`	“Global directive” — lets you pass configuration options directly in the command
`"daemon off;"`	Tells Nginx *not* to detach and background itself
`CMD [...]`	The command Docker runs when the container starts

🧠 Analogy

Think of Docker as a stage manager that keeps a spotlight on **one performer**.
If the performer (the main process) leaves the stage, the lights go out — the container exits.

By saying **“daemon off;”**, we’re telling Nginx:

“Stay on stage and keep performing until the container stops.”

✅ **In short:**
`daemon off;` = “Run Nginx in the foreground so Docker doesn’t think the container has stopped.”

4. Deep dive into docker run -d -p 8080:80 event-app

Here's the command again:

```
docker run -d -p 8080:80 event-app
```

One-line plain-English: Create and start a container from the `event-app` image, run it in the background, and forward requests from your machine's port **8080** to the container's port **80**.

Now let's unpack every piece, what it really does, why it matters, and useful extras.

Parts of the command (explained in depth)

`docker run`

Starts a new container from a Docker image. If the image `event-app` is not present locally, Docker will try to pull `event-app:latest` from a registry (usually Docker Hub).

`-d` (detached mode)

- Runs the container **in the background** (detached), so your terminal is free immediately.
- Docker prints the container ID and returns you to the shell.
- To see output afterwards you use `docker logs -f <container>` (or `docker logs -f <name>`).
- If you **omit `-d`**, the container runs attached to your terminal and you'll see stdout/stderr live (useful for debugging).

`-p 8080:80` (port mapping / publish)

This is the key for making the service inside the container reachable from your host:

- Format: `-p [HOST_IP:]HOST_PORT:CONTAINER_PORT[/PROTOCOL]`.
- `8080:80` means **host port 8080 → container port 80**.
    - When you open `http://localhost:8080` in your browser, the traffic is forwarded to port 80 **inside the container**.
- By default the host side is bound to **0.0.0.0** (all interfaces). You can restrict it to localhost only:
    - `-p 127.0.0.1:8080:80` — only accessible from the machine itself.

- `-P` (uppercase) is different: it publishes **all EXPOSEd** ports from the image to random high-numbered host ports.
- You can publish multiple ports: `-p 8080:80 -p 8443:443`.
- Protocol can be specified: `-p 8080:80/tcp` or `-p 5000:5000/udp`.

Common gotchas:

- If port `8080` on the host is already in use, the `docker run` will fail with "port is already allocated".
- Container must actually listen on the container port (80) — if the process listens only on 127.0.0.1 inside container, the mapping won't work (more on that below).

`event-app`

- The image name. Docker will run the container from this image.
- If you built it locally as `event-app` (default tag `latest`), Docker uses that. If not present, Docker tries `event-app:latest` from a registry.
- You can use a specific tag: `event-app:1.0` or a fully qualified registry name `myrepo/event-app:tag`.

What happens under the hood (networking basics)

- Docker creates the container with its own network namespace and private IP (e.g., `172.17.x.x` on the default bridge).
- Docker sets up NAT/forwarding rules so traffic arriving at the host port (8080) is forwarded into the container IP:port (80).
  - On Linux this is typically done with iptables rules.
  - On macOS/Windows (Docker Desktop) Docker runs a lightweight VM and forwards the traffic from your machine to the VM and then into the container — but from your point of view `http://localhost:8080` works the same.
- Inside the container, the app (Nginx in your image) must be listening on the container interface (usually `0.0.0.0:80`) to accept forwarded traffic.

Inspecting & managing the container

Useful commands you'll want after running it:

- `docker ps` — list running containers and the `PORTS` column (shows `0.0.0.0:8080->80/tcp`).

- `docker port <container>` — show host port(s) mapped to container ports.
- `docker inspect <container>` — raw JSON with `.NetworkSettings.Ports` showing mapping.
- `docker logs -f <container>` — follow container stdout/stderr.
- `docker stop <container>` — stop the container (graceful).
- `docker rm <container>` — remove the stopped container.
- `docker exec -it <container> sh` — open an interactive shell inside the running container (useful for debugging; `sh` works for `alpine`).

Example: name the container so commands are easier:

```
docker run -d --name event --restart unless-stopped -p 
127.0.0.1:8080:80 event-app
```

- `--name event` → refer to the container as `event` instead of the auto-generated id/name.
- `--restart unless-stopped` → docker will restart it automatically on daemon restart or crash (helpful for keeping services up).

Important practical details & gotchas

- **`EXPOSE` in Dockerfile is only documentation.** `EXPOSE 80` does not publish the port to the host — you still need `-p` to make it reachable externally.
- **Binding**: binding to `0.0.0.0` exposes the service to other machines on your network. Use `127.0.0.1:8080:80` to restrict to local access.
- **Privileged host ports (Linux)**: binding to host ports <1024 may require root privileges; `docker` typically runs with sufficient privileges, but keep awareness.
- **Multiple containers and conflicts**: only one container can bind a specific host port at a time. If you need many containers listening on port 80 internally, publish them to different host ports.
- **Container process (PID 1)**: Docker considers the container "running" while its main process (PID 1) runs. That's why `nginx -g "daemon off;"` is used — Nginx runs in the foreground so the container stays alive.
- **Security**: exposing containers to the public internet requires thought (TLS, firewall, reverse proxy). Don't expose services unnecessarily.

Quick examples & useful variants

Start and name container, accessible only locally, auto-restart:

```
docker run -d --name event --restart unless-stopped -p
127.0.0.1:8080:80 event-app
```

Run interactively (foreground) for debugging:

```
docker run -it --rm -p 8080:80 event-app
# --rm removes the container when it exits; -it attaches a
TTY
```

Publish all EXPOSEd ports to random host ports:

```
docker run -P event-app
# then `docker ps` shows which random host ports were
assigned
```

Tail logs of the background container:

```
docker logs -f event
```

Open a shell inside the running container:

```
docker exec -it event sh
# then check listening ports inside: netstat -tuln (if
available) or ss -ltn
```

TL;DR — Key takeaways

- `docker run` creates and starts a container.
- `-d` runs it **detached** (background).
- `-p 8080:80` **publishes** container port `80` on your host at port `8080` so you can access it via `http://localhost:8080`.
- The container must be listening on the container port (80) and Docker must be able to bind the host port (8080).
- Use `--name`, `--restart`, and `-p 127.0.0.1:...` for convenience, persistence and security.