Part 1: Introduction — What Is a Docker Image?

A **Docker image** is (more than) a **blueprint** for a container. Include the actual state executed more than the blueprint. It's a **read-only, layered filesystem** built using a **Dockerfile**.

When you run an image with `docker run`, Docker creates a **container** — a running instance with a writable layer on top of the image.

Part 2: Dockerfile Fundamentals

A `Dockerfile` is a **recipe** that defines how your image is built.

Here's a minimal example:

```
# Simple Dockerfile Example
FROM python:3.12-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

Let's break down **each instruction**.

---

1. `FROM` — Base Image

Syntax:

```
FROM <image>[:<tag>]
```

- It specifies the **base image** (starting point).
- Every image must start with a `FROM` (except for scratch images).

  ARG is the only command that can be before FROM. But that is rarely used.

Examples:

```
FROM ubuntu:22.04
FROM python:3.12-slim
FROM node:20-alpine
```

Best Practices

- Use **lightweight images** (`-alpine`, `-slim`) to reduce image size.
- Always **pin versions** (e.g., `python:3.12-slim`) for reproducibility.
- Use `scratch` for building minimal images (for Go, Rust, etc.).

---

2. `RUN` — Execute Commands During Build

Syntax:

```
RUN <command>
RUN ["executable", "param1", "param2"]
```

Used to install software, set up environment, or configure files — it creates a **new image layer**.

Example:

```
RUN apt-get update && apt-get install -y curl
RUN pip install flask
```

Best Practices

- Combine related commands to reduce image layers:

  ```
  RUN apt-get update && apt-get install -y curl python3-pip && rm -rf /var/lib/apt/lists/*
  ```

- Always **clean up caches** and temporary files.
- Use **multi-stage builds** for smaller final images.

---

3. `CMD` — Default Command to Run in Container

Syntax:

```
CMD ["executable", "param1", "param2"]  # exec form (recommended)
CMD command param1 param2               # shell form
```

- Defines the **default command** that runs when the container starts.
- You can **override it** using `docker run <image> <your_command>`.

Example:

```
CMD ["python", "app.py"]
```

Best Practices

- Always use **JSON (exec) form** — avoids spawning an extra shell.
- Only **one CMD** per Dockerfile — the last one overrides others.

---

4. `ENTRYPOINT` — Fixed Main Command

Syntax:

```
ENTRYPOINT ["executable", "param1", "param2"]
```

- Used to define the **main application** that should always run.
- CMD is often used to provide **default arguments** to ENTRYPOINT.

Example:

```
ENTRYPOINT ["python", "app.py"]
```

Now even if you run:

```
docker run myapp arg1
```

it executes:

```
python app.py arg1
```

Best Practice
Use `ENTRYPOINT` for your main app and `CMD` for its arguments.

---

5. Combining `ENTRYPOINT` and `CMD`

You can use both together for flexibility.

Example:

```
FROM ubuntu:22.04
ENTRYPOINT ["echo"]
CMD ["Hello, World!"]
```

If you run:

```
docker run myimage
```

Output → `Hello, World!`

If you override CMD:

```
docker run myimage Goodbye
```

Output → `Goodbye`

Concept:

- `ENTRYPOINT` = Fixed executable.
- `CMD` = Default argument.

---

6. `ADD` and `COPY` — Copying Files into the Image

Both copy files from **host → image**, but they differ slightly.

`COPY`

Simplest and most common — just copies files/directories.

```
COPY requirements.txt /app/
COPY . /app
```

Best Practice: Use `COPY` whenever possible — it's predictable.

`ADD`

Adds files **and** supports:

- **Remote URLs**
- **Automatic tar extraction**

```
ADD https://example.com/file.tar.gz /tmp/
ADD myapp.tar.gz /app/
```

Best Practice: Use only when you need those extra features.
Don't use `ADD` for normal local file copies — it's less transparent.

---

7. `ENV` — Set Environment Variables

Syntax:

```
ENV <key>=<value>
```

Example:

```
ENV APP_ENV=production
ENV PORT=8080
```

Used to configure runtime environment for your app.

Best Practices

- Use ENV for config that shouldn't change often.
- Use `docker run -e KEY=value` for dynamic values.

Part 3: Building a Real-World Docker Image Example

Let's build a **Flask web app** image — production ready.

Directory structure:

```
flask-app/
├── app.py
├── requirements.txt
└── Dockerfile
```

`app.py`

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello from Docker!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

`requirements.txt`

```
flask==3.0.2
```

`Dockerfile`

```
# Stage 1: Base
FROM python:3.12-slim

# Set working directory
WORKDIR /app

# Copy dependency file first for caching
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the source code
COPY . .

# Set environment variables
ENV PORT=8080

# Expose port
EXPOSE 8080

# Use entrypoint and cmd
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Build and Run

```
docker build -t flask-app:latest .
docker run -p 8080:8080 flask-app
```

Access at → http://localhost:8080 ↗

Part 4: Docker Image Best Practices

Category	Best Practice	Why
Base Image	Use minimal images (`-alpine`, `-slim`)	Reduces size and attack surface
Layers	Combine related RUNs	Fewer layers = smaller image

Caching	Copy dependency files early	Avoids reinstalling packages unnecessarily
Cleanup	Remove caches (`rm -rf /var/lib/apt/lists/*`)	Prevents bloat
Security	Use non-root user (`USER appuser`)	Prevents privilege escalation
Scanning	Use `docker scan` or Trivy	Detects vulnerabilities
Secrets	Don't hardcode passwords or tokens	Use environment variables or Docker secrets
Versioning	Pin base images and dependencies	Ensures reproducibility
Multi-stage Builds	Use multiple stages to reduce size	Only final stage goes to production
Healthcheck	Add `HEALTHCHECK` instruction	Detect and auto-restart unhealthy containers

---

Example — Secure Image Setup

```
FROM python:3.12-slim
WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt && \
    useradd -m appuser

COPY . .
USER appuser
EXPOSE 8080
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Runs as non-root
Minimal image
Cached dependency installation

Part 5: Bonus — Inspecting and Analyzing Your Image

- **List image layers:**

  ```
  docker history flask-app
  ```

- **Inspect metadata:**

  ```
  docker inspect flask-app
  ```

- **Scan for vulnerabilities:**

  ```
  docker scan flask-app
  ```

  or use **Trivy**:

  ```
  trivy image flask-app
  ```

Detailed refernce:

1. Refer to my repository in
   a. https://github.com/vilasvarghese/docker-k8s/dockerfiles
2. Command to use
   a. docker build -t <image name in lower case> -f <file name> .