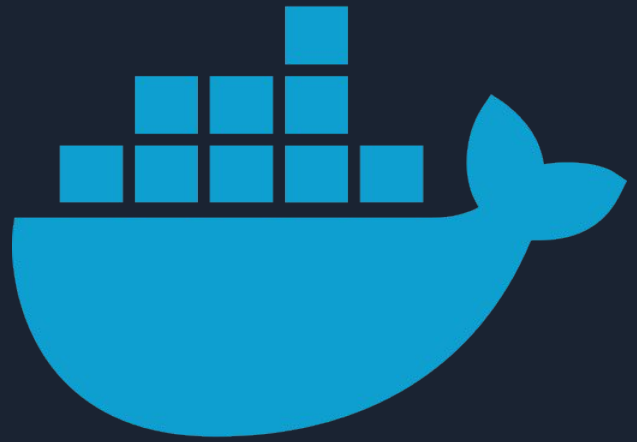
# Docker 101

*Containers from the ground up*

UMBC Linux User's Group – Alex Merryman



# "It works on my machine"

**Developer's Machine**

- Python 3.11
- libssl 1.1
- Ubuntu 22.04

*deploy →*

**Production Server**

- Python 3.8
- libssl 3.0
- CentOS 7

*Docker solves this by packaging your app AND its environment together.*

# What is Docker?

Docker is a tool that lets you run applications inside isolated, portable environments called containers.

### Portable

Build once, run anywhere; your laptop, a server, the cloud.

### Isolated

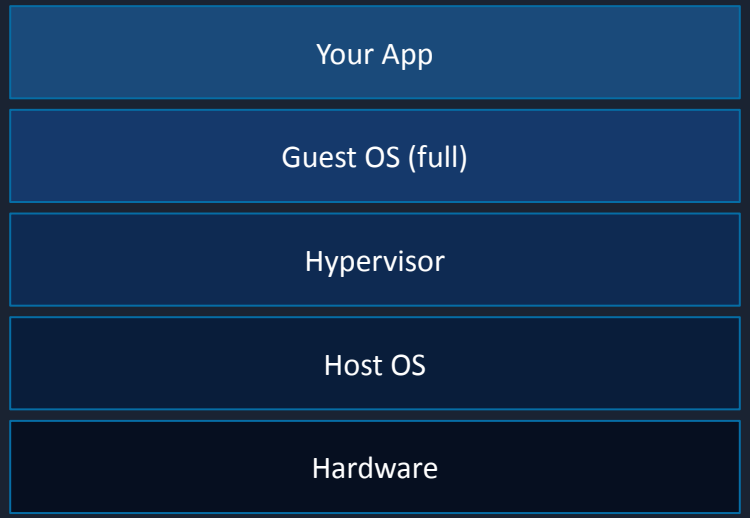
Each container has its own environment. No more dependency conflicts.

### Consistent

Dev, staging, and production all run the exact same environment.

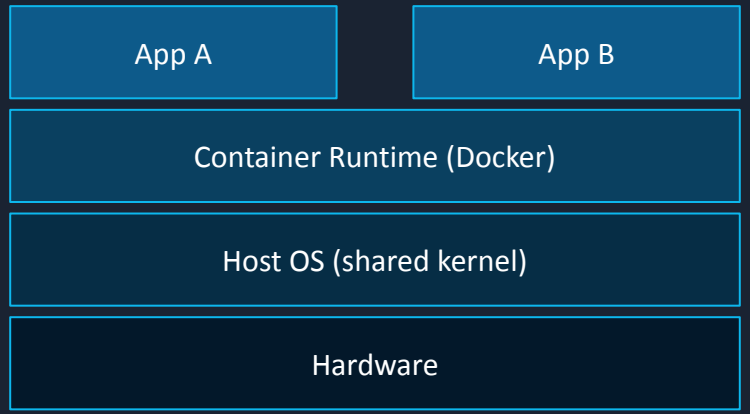
# Containers vs. Virtual Machines

## Virtual Machine




*Full OS = GBs in size, minutes to boot*

## Container




*Shares host OS = MBs in size, starts in seconds*

# Why does anyone use Docker?

## Dev Environments

Share a docker-compose.yml and everyone gets the same setup - no "install Node 18 first" instructions.

## Deploying Apps

Ship containers to any server or cloud provider. No config drift between environments.

## CI/CD Pipelines

Run tests in a clean container on every commit. No stale state between builds.

## Running Services

Spin up Postgres, Redis, or Nginx locally in seconds — no install required.

# 02

# Core Concepts

*Images - Containers - Registries - Layers*

# Images vs. Containers

## Image

- Read-only snapshot
- Built once, reused many times
- Stored in a registry

→

*run*

## Container

- A running image
- Has its own process space
- Ephemeral by default

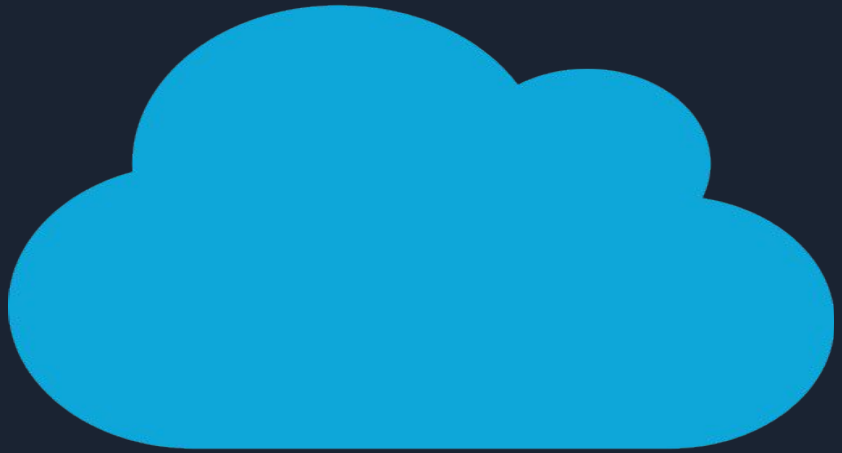
# Docker Hub & Registries

A registry is a place to store and share images.
Docker Hub is the default public registry.

*Anatomy of an image name:*

`nginx:1.28.2-trixie`

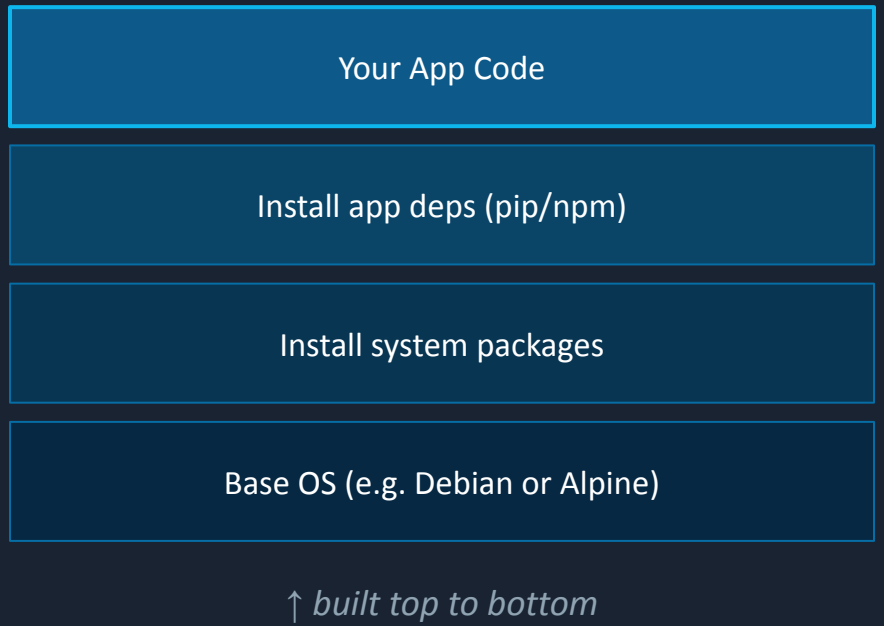
**nginx** — the image name
**1.28.2** — version tag
**trixie** — variant



*Popular images on Docker Hub:*

| nginx | postgres | mysql | node | python | ubuntu |
|---|---|---|---|---|---|

# How Images are Built: Layers




### Cached & Reused

Unchanged layers are cached. Rebuilds are fast - only changed layers re-run.

### Shared

Two images using the same base layer - shared on the disk meaning no duplication

### Why it matters

This is why `docker pull` shows multiple progress bars, one for each layer.

# 03 Running Containers

*Feel free to follow along in your terminal!*

# Your First Container

**1** **Pull an image from Docker Hub**

```
docker pull hello-world
```

**2** **Run it**

```
docker run hello-world
```

If you skip docker pull, docker run will pull automatically. You'll see each layer downloading.

# docker run - Useful Flags

| | |
|---|---|
| `-d` | Detached mode - run in background (you get your terminal back) |
| `-p 8080:80` | Port mapping - host port 8080 → container port 80 |
| `--name web` | Give the container a friendly name instead of a random one |
| `-it` | Interactive + TTY - attach a shell to the container |
| `--rm` | Auto-remove the container when it exits (great for one-off tasks) |

# Run nginx

```
docker run -d -p 8080:80 --name my-nginx nginx
```

*Then open your browser → http://localhost:8080*

**-d**

runs in background

**-p 8080:80**

maps your port 8080 to nginx's port 80

**--name**

gives it a name we can reference later

**nginx**

the image to use (pulled from Docker Hub)

# Managing Containers

```
docker ps
```

List running containers

```
docker ps -a
```

List ALL containers (including stopped ones)

```
docker stop my-nginx
```

Stop a running container gracefully

```
docker start my-nginx
```

Start a stopped container again

```
docker rm my-nginx
```

Delete a stopped container

```
docker logs my-nginx
```

View output/logs from a container

# Managing Images

| | |
|---|---|
| `docker images` | List all images stored locally |
| `docker pull nginx` | Download an image without running it |
| `docker rmi nginx` | Remove a local image (must stop containers first) |
| `docker image prune` | Clean up unused/dangling images to free disk space |

Note: docker system prune cleans up containers, images, networks, and cache all at once.

# 04

# Dockerfiles

*The framework for making custom Docker images*

# What is a Dockerfile?

A Dockerfile is a plain text file with instructions for building a custom image.
Think of it as a script that Docker runs top-to-bottom to set up your environment.

Dockerfile

```
# Start from an official base image
FROM python:3.10-slim-trixie

# Set the working directory
WORKDIR /app

# Copy files into the image
COPY requirements.txt .

# Install dependencies
RUN pip install -r requirements.txt

# Copy the rest of the app
COPY . .

# Default command to run
CMD ["python", "app.py"]
```

**FROM**
Base image to build on top of. Always the first instruction.

**WORKDIR**
Sets the working directory inside the container.

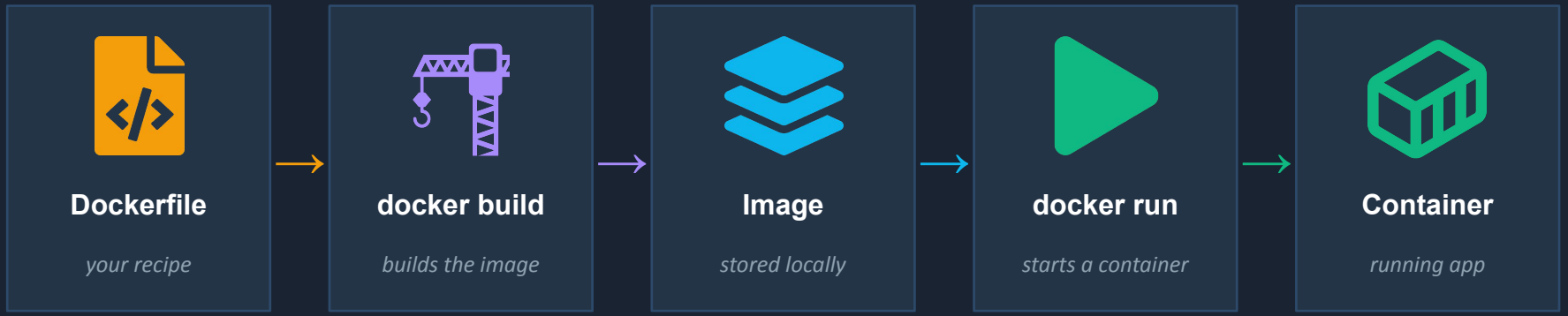
**COPY**
Copies files from your machine into the image.

**RUN**
Runs a shell command during the image build.

**CMD**
The default command to run when the container starts.

SECTION 4 — DOCKERFILES

# From Dockerfile → Running Container




```
docker build -t my-app:latest .     # -t gives it a name/tag, . = build context (current dir)
```

*You won't need to write Dockerfiles today - but you'll see them in almost every project.*

# 05

# Docker Compose

*Orchestrating multiple containers*

# Why Docker Compose?

**Without Compose: running a web app + database:**

```
docker run -d --name db -e POSTGRES_PASSWORD=secret postgres
docker run -d --name web -p 3000:3000 --link db my-app
```

↓ *With Compose, this becomes one file + one command*

```
docker compose up
```

*Define all your services, networks, and volumes in a single docker-compose.yml file.*

# Anatomy of docker-compose.yml

docker-compose.yml

```yaml
services:
  web:
    image: nginx
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
    depends_on:
      - db

  db:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data

volumes:
  db-data:
```

**services**
Top-level key. Each child is a container.

**web / db**
Service names - used to reference each container.

**ports**
Port mapping, same as -p in docker run.

**volumes**
Mount files/folders into the container.

**depends_on**
Ensures 'db' starts before 'web'.

**environment**
Set environment variables inside the container.

```yaml
services:
  app:
    image: 'jc21/nginx-proxy-manager:latest'
    restart: unless-stopped
    ports:
      - '80:80' # Public HTTP Port
      - '443:443' # Public HTTPS Port
      - '81:81' # Admin Web Port
    environment:
      TZ: "Australia/Brisbane"
      # Mysql/Maria connection parameters:
      DB_MYSQL_HOST: "db"
      DB_MYSQL_PORT: 3306
      DB_MYSQL_USER: "npm"
      DB_MYSQL_PASSWORD: "npm"
      DB_MYSQL_NAME: "npm"

    volumes:
      - ./data:/data
      - ./letsencrypt:/etc/letsencrypt
    depends_on:
      - db

  db:
    image: 'jc21/mariadb-aria:latest'
    restart: unless-stopped
    environment:
      MYSQL_ROOT_PASSWORD: 'npm'
      MYSQL_DATABASE: 'npm'
      MYSQL_USER: 'npm'
      MYSQL_PASSWORD: 'npm'
      MARIADB_AUTO_UPGRADE: '1'
    volumes:
      - ./mysql:/var/lib/mysql
```

# NPM

**Then run:**

*Start everything*

```
docker compose up -d
```

*See running services*

```
docker compose ps
```

*View logs*

```
docker compose logs -f
```

*Tear it all down*

```
docker compose down
```

# Volumes & Environment Variables

## Volumes

Containers are ephemeral — when they stop, data is gone.
Volumes persist data beyond the container's lifetime.

```
volumes:
  - db-data:/var/lib/postgresql/data
  - ./html:/usr/share/nginx/html
```

**Named volumes**
db-data:/path — Docker manages storage

**Bind mounts**
./html:/path — maps a local folder


## Env Vars

Pass configuration into containers without hardcoding it in your image.

```
environment:
  POSTGRES_USER: myuser
  POSTGRES_PASSWORD: secret
  POSTGRES_DB: myapp
```

Use a .env file to avoid committing secrets to git.
docker compose picks it up automatically.

```
POSTGRES_PASSWORD=secret   # .env file
```

# 06
# Tips & Best Practices

*Things to know before you get burned*

# Tips, Gotchas & Best Practices

## Don't run as root

Add a USER instruction in your Dockerfile. Root in a container has more reach than you'd expect.

## Use .dockerignore

Like .gitignore — exclude node_modules, .git, secrets. Keeps your image small and fast to build.

## Prefer alpine images

nginx:alpine is ~23MB vs nginx at ~187MB. Smaller images = faster pulls and smaller attack surface.

## Containers are ephemeral

Never store important data inside a container. Use volumes for anything you want to persist.

## One process per container

Containers work best running a single process. Don't cram nginx + app + db into one container.

## Never commit secrets

Don't hardcode passwords in your Dockerfile or docker-compose.yml. Use .env files and .gitignore.

# 07

# What's Next?

*Q&A and further reading*

# Keep Learning

## Docker Docs

docs.docker.com - the official reference. The 'Get Started' guide is a good place to start.

## Host Fun Things

awesome-selfhosted.net/platforms/docker.html - expansive resource of docker applications you can self-host!

## awesome-compose

github.com/docker/awesome-compose - ready-made Compose examples for dozens of stacks.

## Docker Deep Dives

Look into: multi-stage builds, Docker networking, container security, and Kubernetes next.

# Docker Desktop vs. Engine

## Docker Desktop

- GUI app for macOS and Windows
- Includes Docker Engine, Compose, and a dashboard
- Free for personal use - paid for large companies
- Easiest way to get started on non-Linux machines

## Docker Engine

- The core daemon that runs on Linux
- What's actually running on production servers
- No GUI - fully CLI driven
- Installed directly on Linux (what we used today)

## Podman

- A Docker-compatible alternative
- Rootless by default - better security posture
- Drop-in replacement for most docker commands
- Popular in enterprise / RHEL environments

# Questions?

**You now know:**

- ✓ What Docker is and why it exists
- ✓ Images, containers, layers, and registries
- ✓ How to run and manage containers
- ✓ How to read a Dockerfile
- ✓ How to spin up multi-container apps with Compose