

The ULTIMATE Docker Project

Level 1: Docker Fundamentals

Containers vs VMs - Understanding the core difference
Docker Architecture - Client, daemon, registry
Images & Containers - Building blocks of Docker
Basic Commands - run, build, pull, push
Dockerfile Basics - Creating custom images
Port Mapping - Exposing container services
Volume Mounting - Data persistence

Level 2: Installation & Basic Operations

Docker Installation - Linux, Windows, macOS
Docker Desktop - GUI management tool
Container Lifecycle - Create, start, stop, remove
Image Management - Tagging, versioning, cleanup
Container Networking - Bridge, host, none modes
Environment Variables - Configuration management
Log Management - Viewing and managing logs

Level 4: Production Readiness & Security

Security Best Practices - Non-root users, scanning
Multi-stage Builds - Optimizing image size
Health Checks - Container monitoring
Resource Limits - CPU, memory constraints
Secrets Management - Secure credential handling
Registry Security - Private registries, authentication
Monitoring & Logging - Production observability

Level 3: Multi-Container Applications & Orchestration

Docker Compose - Multi-container applications
Service Dependencies - Container communication
Custom Networks - Isolated container networking
Load Balancing - Distributing traffic
Database Integration - Persistent data services
Docker Swarm - Basic orchestration
Scaling Services - Horizontal scaling patterns

Part Two Covers:

Level 2: Installation and Basic Operations

Part Two Covers:


Level 3: Multi Container Applications & Orchestration

Part Three Covers:

Level 4: Production Readiness & Security + **The Ultimate Docker Project**

Level Two

Installation

We first need to install Docker. I'm going to be doing on this on MacOS, it will almost identical to Linux process. If you're using Windows you install WSL, i've written a guide [Here](#) 

Step 1: Install Homebrew (if not already installed)

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Step 2: Install Docker

```
brew install --cask docker
```

Step

3: Start Docker Desktop

```
open /Applications/Docker.app
```


Wait

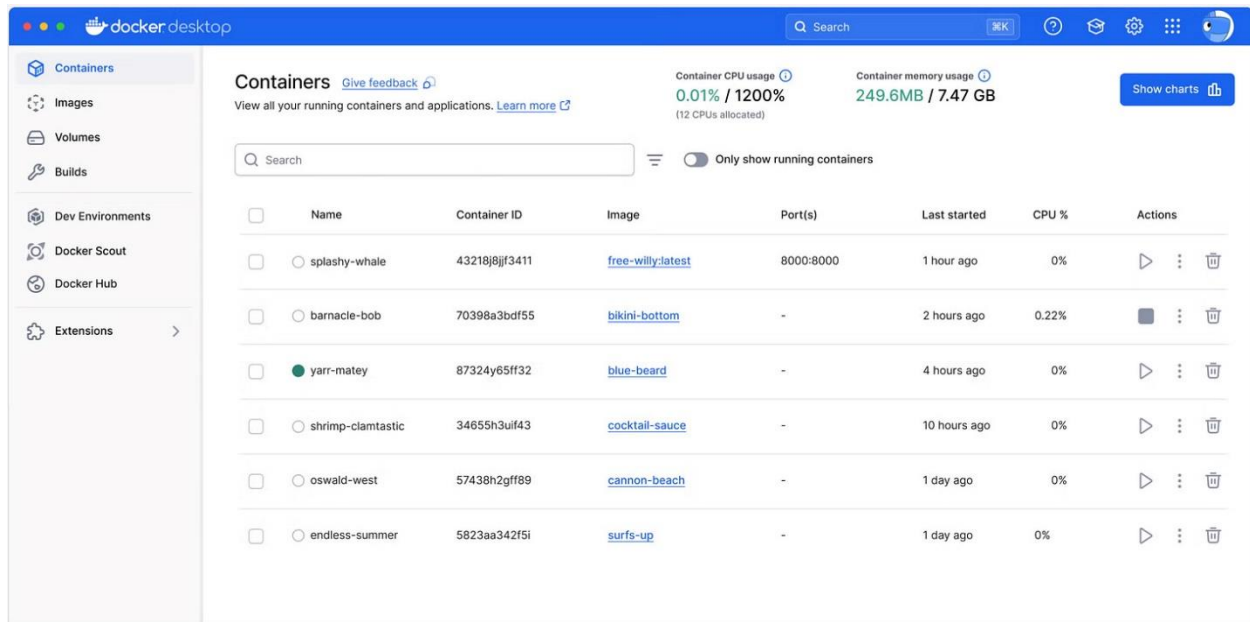
for Docker to finish starting (the whale icon in your status bar should stop animating).

Double check it's installed with

```
Last login: Sat Jul 19 18:20:31 on ttys001  
williampearce@Williams-MacBook-Air ~ % docker --version  
Docker version 27.3.1, build ce12230  
williampearce@Williams-MacBook-Air ~ % |
```

Docker Desktop

[Here](#)  is a link to the steps to install Docker Desktop for macOS, Linux, and Window which I also recommend you use when getting started, you can view your Containers, Images, Volumes etc



Basic Operations

Images (Templates for containers):

```
docker pull <image>           # Download an image from Docker Hub
docker images                 # List all downloaded images
docker rmi <image>            # Remove an image
docker build -t <name> .      # Build image from Dockerfile in current directory
```

Containers (Running instances):

```
docker run <image>             # Create and start a container
docker run -it <image>         # Run interactively with terminal
docker run -d <image>          # Run in background (detached)
docker run -p 8080:80 <image>  # Map host port 8080 to container port 80
docker run --name <name> <image> # Give container a custom name

docker ps                     # List running containers
docker ps -a                  # List all containers (including stopped)
docker stop <container>       # Stop a running container
docker start <container>      # Start a stopped container
docker restart <container>    # Restart a container
docker rm <container>         # Remove a stopped container
```

Container interaction:

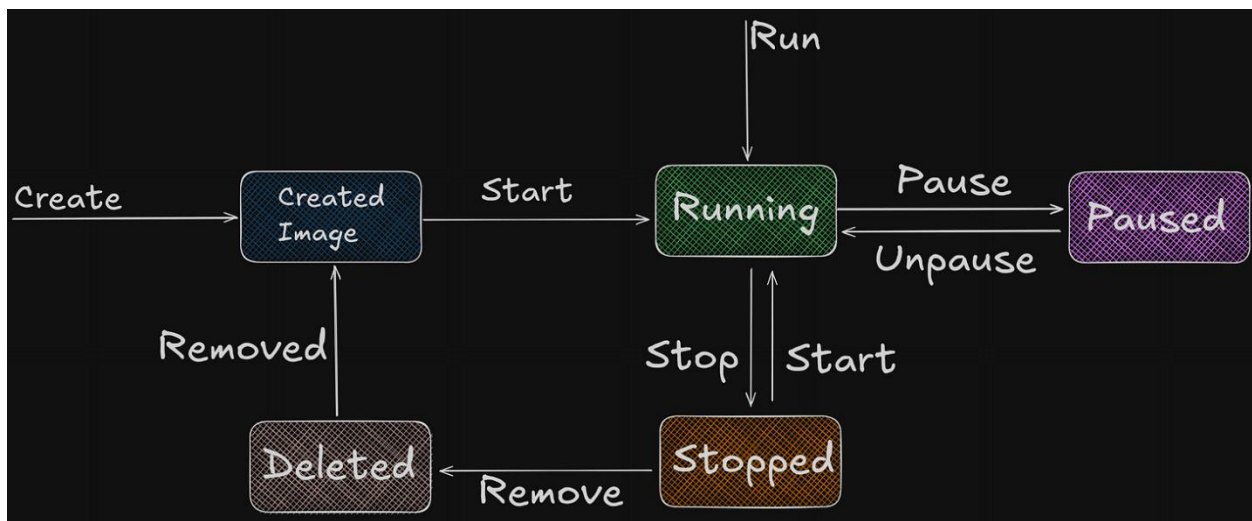
```
docker exec -it <container> bash    # Open bash shell in running container
docker logs <container>              # View container logs
docker cp <file> <container>:/path  # Copy file to container
```

With

Docker installed and a list of basic commands we need to get started let's take a look at we refer to as the “Container Lifecycle”

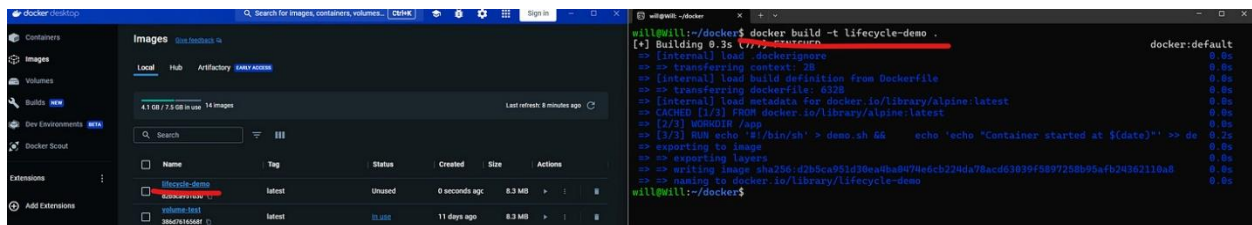
Container Lifecycle

The Journey of a Container can be broken down into six key part



Follow along here if you like:

1. Image: The blueprint



2. **Create:** Docker makes a container from the image

```
will@Will: ~/docker
will@Will:~/docker$ docker create --name demo-container lifecycle-demo
e7188ab06af0fe3d5f7b1ce249ad3ba4d0874826e099bb34157d4ffe852c2ea5
will@Will:~/docker$ s
```

3. **Start:** Container begins running

Container CPU usage ⓘ
0.00% / 1000% (10 cores available)

Container memory usage ⓘ
484KB / 7.41GB

Show charts ▾

☐ Only show running containers

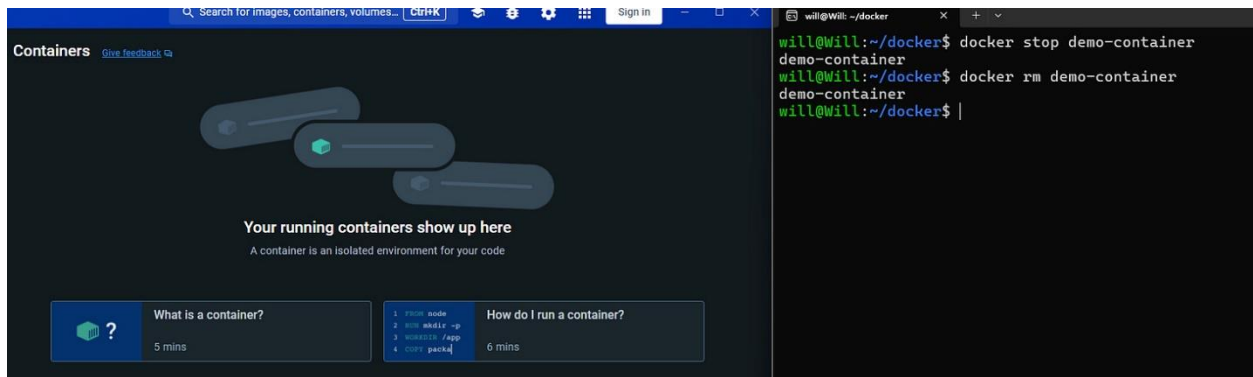
<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	demo-container e7188ab06af0	lifecycle-demo	Running	0%		56 seconds ago	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

```
will@Will:~/docker$ docker start demo-container
demo-container
will@Will:~/docker$ |
```

4. **Running:** Container is actively doing work


```
will@Will:~/docker$ docker logs demo-container # See the work being done
Container started at Thu Jul 24 07:47:15 UTC 2025
Container ID: e7188ab06af0
Waiting... Press Ctrl+C to stop
Still running at Thu Jul 24 07:47:15 UTC 2025
Still running at Thu Jul 24 07:47:25 UTC 2025
Still running at Thu Jul 24 07:47:35 UTC 2025
Still running at Thu Jul 24 07:47:45 UTC 2025
Still running at Thu Jul 24 07:47:55 UTC 2025
Still running at Thu Jul 24 07:48:05 UTC 2025
Still running at Thu Jul 24 07:48:15 UTC 2025
Still running at Thu Jul 24 07:48:25 UTC 2025
Still running at Thu Jul 24 07:48:35 UTC 2025
will@Will:~/docker$ |
```

5. **Stop:** Container pauses but data remains
6. **Remove:** Container is deleted permanently



Think of it like this:

- **Image** = Recipe
- **Container** = Cake made from recipe
- **Running** = People eating the cake
- **Stopped** = Cake sitting on counter
- **Removed** = Cake thrown away

Important: When you remove a container, any data inside it is lost unless you use volumes to save it, check [part one here](#)  for more on volumes

Most beginners just use `docker run` (which creates and starts) then `docker stop` and `docker rm` when done.

Image Management

When you start using Docker it's pretty simple to manage what containers you have locally but in an organisation (or even a large homelab) things can get messy pretty quickly. That's why proper image management is essential. This is done through:

Tagging

```
docker tag myapp:latest myapp:v1.2.3      # Create version tag
docker tag myapp:latest myregistry.com/myapp # Tag for registry
docker build -t myapp:production .         # Tag during build
```


Tags help you identify different versions and environments (dev, staging, prod).

See how it looks like its created a new image, you're not creating a new image; you're pointing a new tag (v1.2.3) to the same underlying image as latest.

```
will@will:~/docker$ docker tag substack-image:latest myapp:v1.2.3
will@will:~/docker$ docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
myapp	v1.2.3	3a78443545b4	24 seconds ago	8.31MB
substack-image	latest	3a78443545b4	24 seconds ago	8.31MB

Versioning

```
docker pull nginx:1.21      # Specific version
docker pull nginx:latest    # Latest version (default)
docker pull nginx:alpine    # Variant (smaller, Alpine-based)
```

Much the same as tags but when pulling images, always use specific versions in production instead of latest to avoid surprises. Maybe a specific versions is known to have vulnerabilities

Docker Registry:

We haven't really spoken about registries, I did think about including it in part one but wanted to lay the groundwork first.

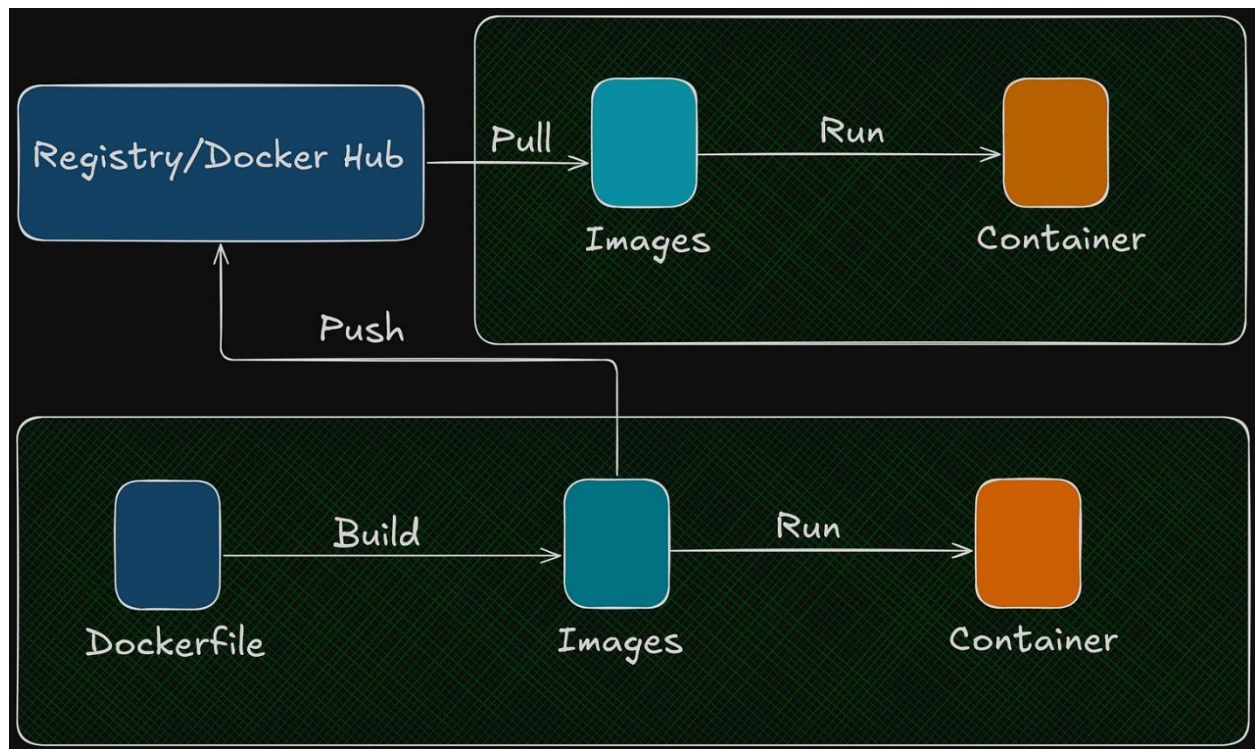
```
docker push myregistry.com/myapp:v1.2.3    # Upload to registry
docker pull myregistry.com/myapp:v1.2.3    # Download from registry
```

Once

you've built your Docker image, you'll likely want to store it somewhere safe and accessible, that's where Docker registries come in.

Think of a registry like a cloud based library for your images. Docker Hub is the most common one, but platforms like GitHub Container Registry or AWS ECR are also popular. You can "push" your image to the registry using simple commands, then "pull" it back down on any machine or server.

This makes it easy to share your work, deploy to production, or just back things up. It's also how teams collaborate, tagging versions like v1.0.0 helps everyone stay on the same page.



Cleanup:

```
docker system prune      # Remove unused containers, networks, images
docker container prune   # Remove all stopped containers
docker image prune       # Remove unused images
docker image prune -a    # Remove ALL unused images (not just dangling)
```

Container Networking

Let's keep this simple, like anything with networking the complexity comes as scale but the fundamentals remain the same.

Right, container networking basics. Here are the key fundamentals for beginners.

Things you should know:

- Containers automatically join Docker's default bridge network
- They get internal IP addresses (like 172.17.0.2)
- Containers can talk to each other using these IPs

- -p host_port:container_port exposes container services
- Example: -p 8080:80 maps host port 8080 to container port 80

Container Name Resolution

- Containers can find each other by name on custom networks
- docker network create mynetwork creates isolated networks
- --network mynetwork connects containers to it

Key Commands

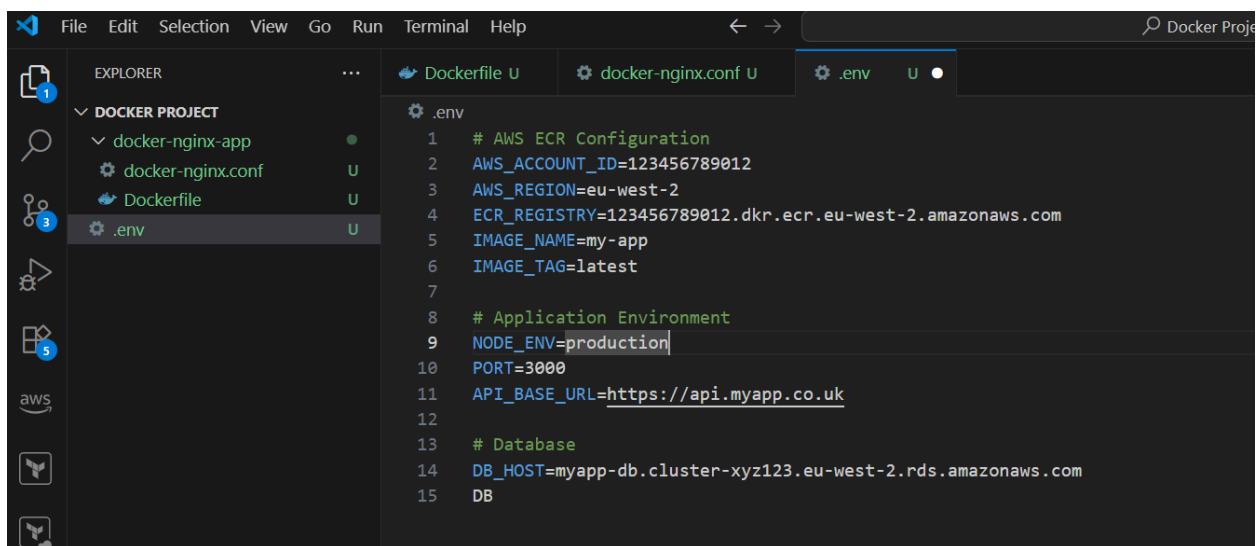
- docker network ls - list networks
- docker network inspect bridge - see network details
- docker port container_name - check port mappings

The main point: containers are isolated by default but can communicate through networks and port mapping.

I'm going to dive deeper into this in the final project part 😊

Environment Variables

Environment variables let you pass bits of configuration into your Docker containers without hardcoding them into your app.



The screenshot shows a Visual Studio Code editor window with a project named 'DOCKER PROJECT'. The Explorer sidebar on the left shows the file structure with files like 'docker-nginx-app', 'docker-nginx.conf', 'Dockerfile', and '.env'. The '.env' file is selected and its contents are displayed in the main editor. The file contains configuration for AWS ECR and application environment variables.

```
.env
1 # AWS ECR Configuration
2 AWS_ACCOUNT_ID=123456789012
3 AWS_REGION=eu-west-2
4 ECR_REGISTRY=123456789012.dkr.ecr.eu-west-2.amazonaws.com
5 IMAGE_NAME=my-app
6 IMAGE_TAG=latest
7
8 # Application Environment
9 NODE_ENV=production
10 PORT=3000
11 API_BASE_URL=https://api.myapp.co.uk
12
13 # Database
14 DB_HOST=myapp-db.cluster-xyz123.eu-west-2.rds.amazonaws.com
15 DB
```

Think things like API keys, database passwords, or different settings for dev and production. Instead of baking these values into your image, you can supply them when you run the container using the `-e` flag (`-e PORT=3000`). This keeps your images cleaner, more secure, and easier to reuse.

You can also load lots of variables at once from a `.env` file, this is way more common than passing them in at run time

DON'T PUSH THIS FILE TO GITHUB

Or do, it keeps me in the job 😊

Log Management

Container log management in Docker is straightforward once you understand the basics.

Docker automatically captures anything your application writes to `stdout` or `stderr` and stores it using the `json-file` driver by default. You can view these logs using

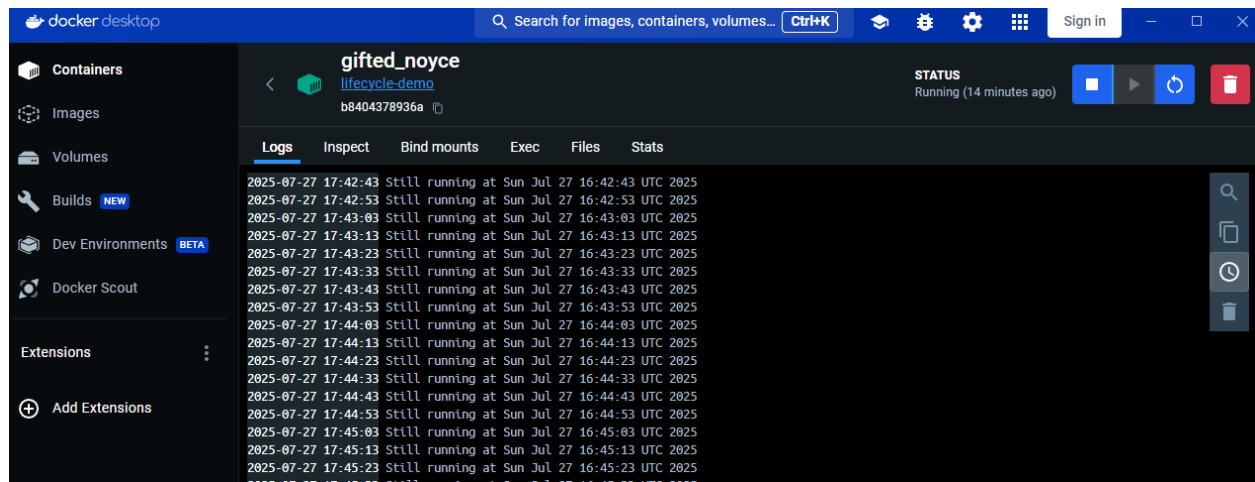
```
docker logs container_id
```

add `-f` to follow them in real-time, or use options like `--tail 50` to see just the last 50 lines.

```
will@will:~$ docker logs b8404378936a
Container started at Sun Jul 27 16:39:23 UTC 2025
Container ID: b8404378936a
Waiting... Press Ctrl+C to stop
Still running at Sun Jul 27 16:39:23 UTC 2025
Still running at Sun Jul 27 16:39:33 UTC 2025
Still running at Sun Jul 27 16:39:43 UTC 2025
Still running at Sun Jul 27 16:39:53 UTC 2025
Still running at Sun Jul 27 16:40:03 UTC 2025
Still running at Sun Jul 27 16:40:13 UTC 2025
Still running at Sun Jul 27 16:40:23 UTC 2025
```

The key thing you need to know is that Docker handles log collection automatically, but without proper configuration, these logs will grow indefinitely and fill up your disk.

Neat trick, you can also view them in Docker Desktop



You can prevent this by setting log rotation limits with `--log-opt max-size=10m --log-opt max-file=3` when running containers, which keeps only the last 3 files of 10MB each.

The golden rule is simple: make sure your applications log to stdout/stderr rather than writing to files inside the container, and Docker will take care of collecting and managing those logs for you.

Are you enjoying these broken-down **"Ultimate Projects"** or do you prefer short and to the point tutorials? I'm trying to find the right balance at the moment. Let me know what you think - we're going to continue in part three and wrap it up with a project in part four.