Container Lifecycle & Debugging — Study Notes

> **Core Principle:** Containerized applications fail silently unless you know where to look.

Table of Contents

1. Container Lifecycle Overview
2. This Project Structure
3. Essential Debugging Commands
4. Common Errors & Fixes
5. Restart Policies
6. Recommended Settings
7. Docker Compose Tips
8. Quick Reference Card

---

1. Container Lifecycle Overview

A Docker container moves through the following states:

```
Created → Running → Paused (optional) → Stopped → Removed
                ↘ Exited (on crash/completion)
```

State	Description
`created`	Container created but not yet started
`running`	Process is executing inside the container
`paused`	All processes frozen (via `docker pause`)
`exited`	Process finished (successfully or with error)
`dead`	Container could not be properly removed
`removing`	Container is being deleted

Key Insight

- A container's lifecycle is tied to its **main process (PID 1)**
- When PID 1 exits, the container exits — regardless of exit code
- Exit code `0` = success; `1` = error; `137` = kernel killed (OOM or `docker kill`)

---

2. This Project Structure

```
FastAPIApplication/
├── Dockerfile          # Image definition
├── compose.yaml        # Multi-container orchestration
├── main.py             # FastAPI application
├── pyproject.toml      # Python project metadata & dependencies
└── uv.lock             # Locked dependency versions
```

Dockerfile — Annotated

```
FROM python:3.13-slim           # Slim base = smaller image, fewer attack surfaces

WORKDIR /app                    # All subsequent commands run from /app

RUN pip install --no-cache-dir uv  # Install uv package manager (--no-cache saves space)

COPY pyproject.toml /app/       # Copy dependency files BEFORE source code
COPY uv.lock /app/              # Enables Docker layer caching for dependencies

RUN uv sync --frozen            # Install exact locked versions (--frozen = no updates)

COPY main.py .                  # Copy source last — cache invalidated only on src change

EXPOSE 8000                     # Documents the port; does NOT publish it to host

LABEL NAME="..." VERSION="..."  # Metadata; searchable via docker inspect

CMD ["uv","run","uvicorn","main:app","--host","0.0.0.0","--port","8000","--reload"]
# --host 0.0.0.0  → bind to ALL interfaces (required inside container)
# --reload        → auto-restart on code changes (DEV only — do NOT use in production)
```

> **Hint:** Ordering `COPY` + `RUN` strategically maximizes Docker's build cache. Dependencies change less often than source code, so copy them first.

compose.yaml — Annotated

```
services:
  api:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8000:8000"             # host_port:container_port
    volumes:
      - .:/app                  # Bind mount: host directory → /app in container
                                # Required for --reload to see host file changes
    environment:
      - DEBUG=true              # Pass env vars at runtime (not baked into image)
    restart: unless-stopped     # Auto-restart on crash; stop only if manually stopped
```

---

3. Essential Debugging Commands

`docker logs` — Read Output

```
# Show all logs
docker logs <container-name>

# Stream logs live (follow mode)
docker logs -f <container-name>

# Show only last N lines
docker logs --tail 20 <container-name>

# Combine: follow + tail
docker logs -f --tail 50 <container-name>
```

> **Hint:** Always check `docker logs` FIRST when a container misbehaves. It captures stdout/stderr: startup messages, tracebacks, request logs.

`docker exec` — Run Commands Inside Container

```
# Run a one-off command
docker exec <container-name> pwd
docker exec <container-name> ls /app

# Interactive shell (for investigation)
docker exec -it <container-name> sh      # Alpine/slim images use sh
docker exec -it <container-name> bash    # Full images use bash

# Test internal API without exposing ports
docker exec <container-name> curl -s http://localhost:8000/
docker exec <container-name> curl -s http://localhost:8000/health
```

> **Warning:** Avoid launching shells in **production** containers unless absolutely necessary. Interactive debugging can affect running services and is a security risk.

`docker inspect` — Full Container Metadata

```
# Full JSON dump of container config
docker inspect <container-name>

# Targeted queries using Go template format
docker inspect --format='{{.State.Status}}' <container-name>
docker inspect --format='{{.State.ExitCode}}' <container-name>
docker inspect --format='{{json .Config.Cmd}}' <container-name>
docker inspect --format='{{json .Config.Env}}' <container-name>
docker inspect --format='{{json .NetworkSettings.Ports}}' <container-name>
```

> **Hint:** Use read-only commands (`logs`, `inspect`) before resorting to interactive `exec` — especially in production.

`docker ps` — List Running Containers

```
# Running containers
docker ps

# All containers (including stopped)
docker ps -a

# Formatted table with names and ports
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"
```

`docker stats` — Live Resource Usage

```
# Live CPU, memory, network stats
docker stats

# Single container
docker stats <container-name>

# One-time snapshot (no stream)
docker stats --no-stream <container-name>
```

---

4. Common Errors & Fixes

Error: Port Already Allocated

```
Error response from daemon: driver failed programming external connectivity:
Bind for 0.0.0.0:8000 failed: port is already allocated
```

Cause: Another container or process is already using host port `8000`.

Fix Option 1 — Map to a different host port:

```
docker run -p 8001:8000 <image-name>
```

Fix Option 2 — Find and stop the conflicting container:

```
docker ps --format "table {{.Names}}\t{{.Ports}}"
docker stop <conflicting-container>
```

Error: Container Exits Immediately (Missing Env Var)

Symptom: Container starts then immediately shows `Exited (1)` in `docker ps -a`.

Diagnosis:

```
docker logs <container-name>          # Check for error messages
docker inspect --format='{{.State.ExitCode}}' <container-name>   # Check exit code
```

Fix: Supply the missing environment variable:

```
docker run -e API_KEY=your_value <image-name>
# or in compose.yaml:
environment:
  - API_KEY=your_value
```

Error: `--reload` Not Picking Up Changes

Symptom: Code changes on host not reflected inside container.

Cause: No volume mount, so the container is running the image's copy of the code.

Fix: Add a bind mount in compose.yaml:

```
volumes:
  - .:/app
```

Error: `Cannot connect to the Docker daemon`

```
# Check if Docker daemon is running
sudo systemctl status docker

# Start if stopped
sudo systemctl start docker
```

Error: `ModuleNotFoundError` or `ImportError` at startup

Cause: Dependencies installed in image don't match what the app needs.

Fix:

```
# Rebuild the image from scratch
docker compose build --no-cache
docker compose up
```

Error: `Connection refused` on localhost

Symptom: `curl http://localhost:8000` fails even though container is running.

Common causes:

1. App bound to `127.0.0.1` instead of `0.0.0.0` inside container
2. Port not mapped: missing `-p 8000:8000`
3. App crashed after starting

Fix: Ensure `--host 0.0.0.0` in the uvicorn command AND `-p 8000:8000` in the run command.

---

5. Restart Policies

Policy	Behavior	Use Case
`no`	Never restart (default)	One-shot tasks
`always`	Always restart, even on manual stop	Avoid (too aggressive)
`unless-stopped`	Restart on crash; stop only if manually stopped	**Production APIs**
`on-failure`	Restart only on non-zero exit code	Batch jobs
`on-failure:3`	Restart on error, maximum 3 times then give up	Fault-tolerant jobs

Recommended for this project: `restart: unless-stopped` (already in compose.yaml)

```
# Set restart policy on existing container
docker update --restart unless-stopped <container-name>

# Verify
docker inspect --format='{{.HostConfig.RestartPolicy.Name}}' <container-name>
```

---

6. Recommended Settings

Development (local)

```
# compose.yaml
services:
  api:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - .:/app                   # Bind mount for hot-reload
    environment:
      - DEBUG=true
    restart: unless-stopped
```

```
# CMD in Dockerfile
CMD ["uv", "run", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
```

Production

```
# Use multiple workers, remove --reload
CMD ["uv", "run", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--workers", "4"]
```

```
# compose.yaml for production
services:
  api:
    image: your-registry/fastapi-app:1.0.0   # Use pre-built image tag
    ports:
      - "8000:8000"
    # NO volumes bind mount in production
    environment:
      - DEBUG=false
    restart: unless-stopped
    deploy:
      resources:
        limits:
          memory: 512M
```

> **Warning:** Never use `--reload` in production. It watches the filesystem and adds overhead; it is a development-only feature.

`.dockerignore` — Recommended

Always create a `.dockerignore` to avoid copying unnecessary files into the image:

```
.venv/
__pycache__/
*.pyc
*.pyo
.git/
.env
.env.*
*.md
tests/
.pytest_cache/
```

---

7. Docker Compose Tips

```
# Build and start in detached mode
docker compose up -d

# Build without cache (fresh build)
docker compose build --no-cache

# View logs for all services
docker compose logs

# Follow logs for specific service
docker compose logs -f api

# Stop and remove containers (keep volumes)
docker compose down

# Stop and remove containers AND volumes
docker compose down -v

# Restart a single service
docker compose restart api

# Execute command in running service
docker compose exec api sh
```

> **Hint:** `docker compose up --build` rebuilds the image before starting. Use this when you change the `Dockerfile` or `pyproject.toml`.

---

8. Quick Reference Card

Debugging Workflow

```
Container not working?
│
├─ 1. docker ps -a                    → Is it running or exited?
│
├─ 2. docker logs <name>              → What did it print? Any errors?
│
├─ 3. docker inspect <name>           → Exit code? Env vars? Ports?
│
├─ 4. docker exec -it <name> sh       → Can you get inside? Is the app running?
│
└─ 5. docker compose build --no-cache → Rebuild if deps changed
```

Exit Codes

Exit Code	Meaning
`0`	Success — clean shutdown
`1`	Application error
`2`	Misuse of shell built-in
`126`	Permission denied
`127`	Command not found
`137`	SIGKILL — OOM or `docker kill`
`143`	SIGTERM — graceful shutdown

Build Optimization Rules

1. **Put rarely-changing layers at the top** (base image, system deps)
2. **Copy dependency files before source code** (`pyproject.toml` before `main.py`)
3. **Use `--no-cache-dir`** with pip to avoid bloating the layer
4. **Use slim base images** (`python:3.13-slim` vs `python:3.13`)
5. **Always use `.dockerignore`** to exclude `.venv/` , `__pycache__/` , `.git/`

This Project's Endpoints

Method	Path	Description
GET	`/`	Root — hello message
GET	`/application`	Application info message
GET	`/docs`	Swagger UI (auto-generated by FastAPI)
GET	`/redoc`	ReDoc UI

Source

- Container Lifecycle and Debugging — Panaversity AI Cloud Native Development