ShipShape DevOps: Intro to Docker

The Professor Solo Official Study Guide

Escaping the Local-Dev Trap

The ‘Works on My Machine’ Problem

The Local-Dev Trap

A Node app feels finished when it runs locally, but local environments are full of invisible assumptions: specific Node versions, lingering **.env** values, and host OS quirks. Relying on the weather is a dangerous way to ship.

Operational Check, Mate

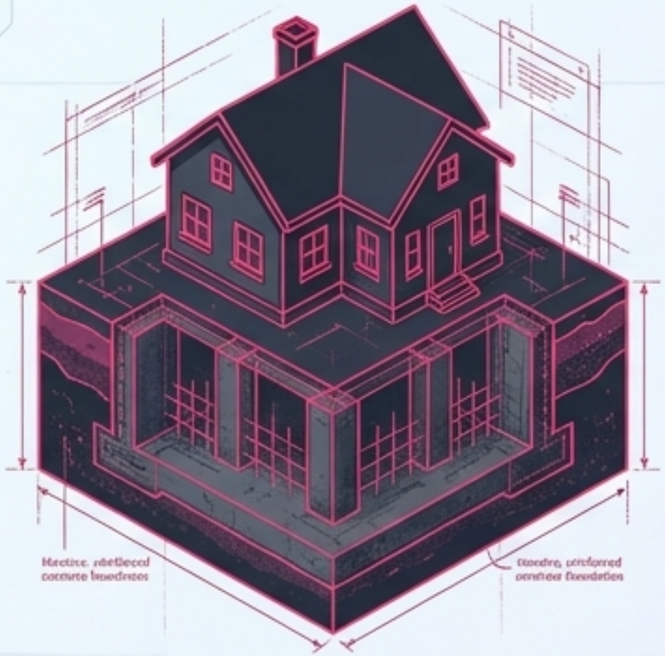
The Docker Promise:

The Docker Promise: We package the runtime with the code. We aren't just shipping the application; we are shipping the exact environment it needs to survive in the wild. Runtime consistency is designed, not hoped for.

Stop relying on the weather. Pack the cargo properly.

Architecture: Virtual Machines vs. Containers

Virtual Machines (The Heavy Haulers)



- Virtualizes the hardware.
- Boots a full Guest OS for every app.
- Massive file sizes (Gigabytes).
- Takes minutes to boot.

Containers (The Standardized Fleet)

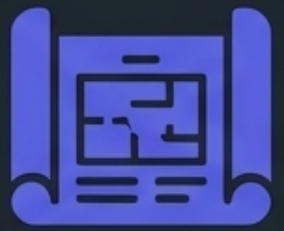
- Virtualizes the operating system.
- Shares the host machine's Linux kernel.
- Lightweight footprint (Megabytes).
- Boots in milliseconds.

Professor Solo Says: A container is an isolated application process, NOT just a 'tiny VM.'

The Docker Bay: Core Pillars

Docker Engine

The shipyard foreman. The underlying software that manages the lifecycle of your builds, networks, and orchestration.



Images

The cold storage blueprint. A packaged, read-only snapshot of your filesystem, OS, and dependencies.

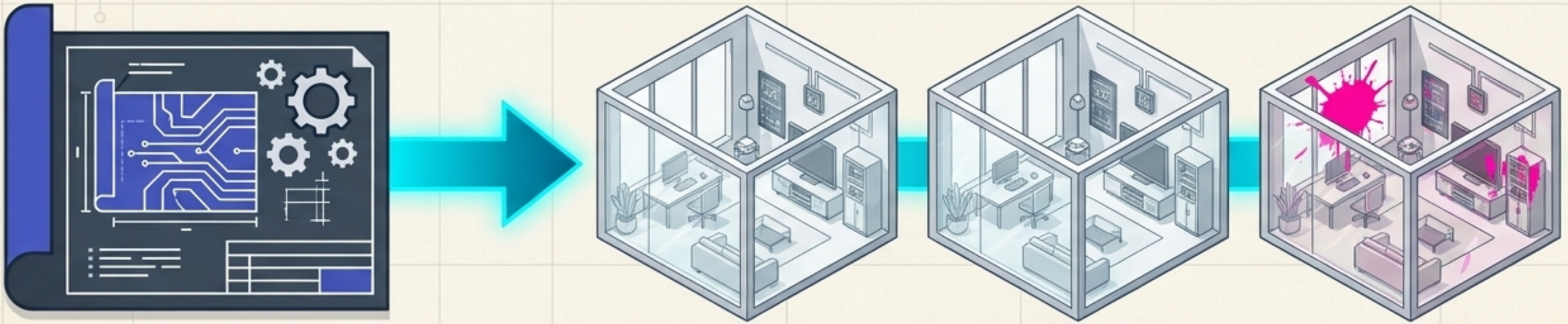
Containers

The live execution. The operational manifestation of an image, running in an isolated environment.

Layers

The secret to speed. Stackable, immutable instruction steps that Docker caches to make rebuilding incredibly fast.

The Golden Rule: Image vs. Container



The Image: The Packaged Truth

- Read-only, immutable artifact.
- Verbs: Build, Tag, Push, Pull.
- If the blueprint changes, everything built from it changes. It is cold storage.

The Container: The Writable Instance

- A live environment with a thin writable layer on top.
- Verbs: Create, Run, Start, Stop, Remove.
- Manual changes disappear when it dies. This is a feature, not a bug! If it breaks, we don't fix it—we kill it and spawn a fresh one from the image.

Professor Solo Says:
Think of an image as a class and a container as an instance.

We BUILD images, and we RUN containers.

The Recipe: Anatomy of a Dockerfile

The Docker Bay Six: Command Reference

FROM	Always prioritize Official verified images over random community ones for security.
COPY	Moves files from host to image. **The trailing '.' refers to the build context, not the parent directory. Space matters!**
EXPOSE	Documentation only! A polite sticky note saying 'I listen on 3000.' It does **NOT** publish ports to the host.
RUN	Happens when the image is built. **You can have many of these instructions.**
CMD	Happens when the container starts. **There can be only one.**

Layer Caching & Architecture

The Standard Pattern (Fast Rebuilds) ✅

The Less Helpful Version (Cache Busted) ⚠

The Physics of Docker

Every instruction creates a discrete, cacheable layer. Docker **reuses layers** until an input changes. A change to line X invalidates the cache for line X and every single line below it.

The Node.js Pattern

Copy the `package.json` and **install dependencies** before copying the source code. If we only change a route file, the dependency layer remains cached, skipping a slow npm install.

The .dockerignore Shield

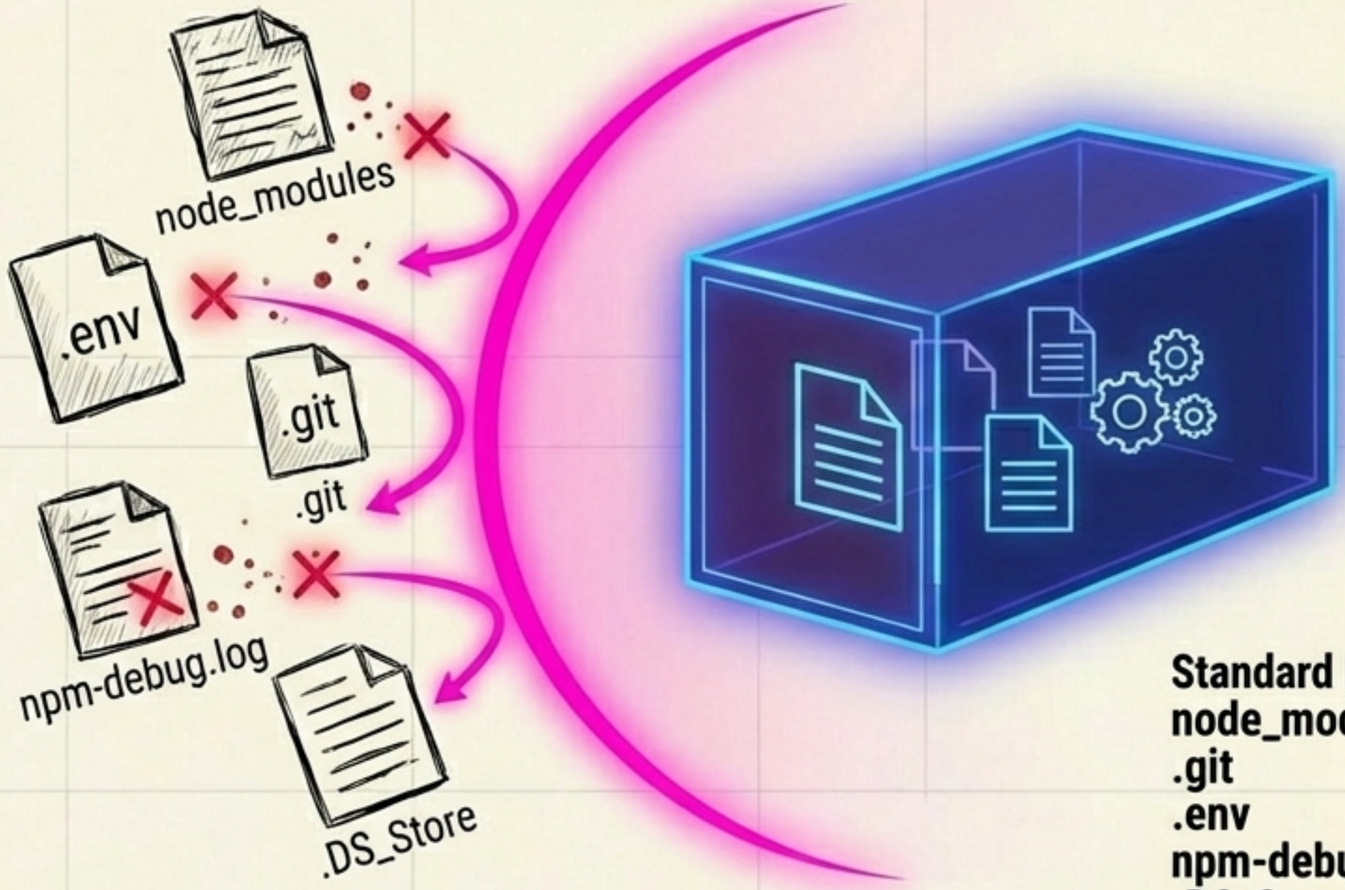
Identical in spirit to **.gitignore**, but for the Docker build daemon.

Host Contamination

Never copy your host laptop's **node_modules** into the final Linux image. Into the final Linux image. It causes massive architectural mismatch errors.

Leaky Secrets

Never copy a local **.env** file containing secret keys into an image blueprint.



Standard Ignored Files:
node_modules
.git
.env
npm-debug.log
.DS_Store

The Result: Faster, smaller, quieter, and infinitely more secure builds.

Operational Workflow: Build then Run

The Build

```
❯ docker build -t my-simple-server .
```

`build`: starts the engine.
`-t`: tags the image with a name.
`.` (dot): mandatory. Tells Docker to use the current directory as the build context.

ⓘ **Note:** The app is NOT running yet. It is just cold storage.

The Run

```
❯ docker run -p 3000:3000 my-simple-server
```

`run`: Spawns a live container from the built image.
`-p`: Maps the network traffic.

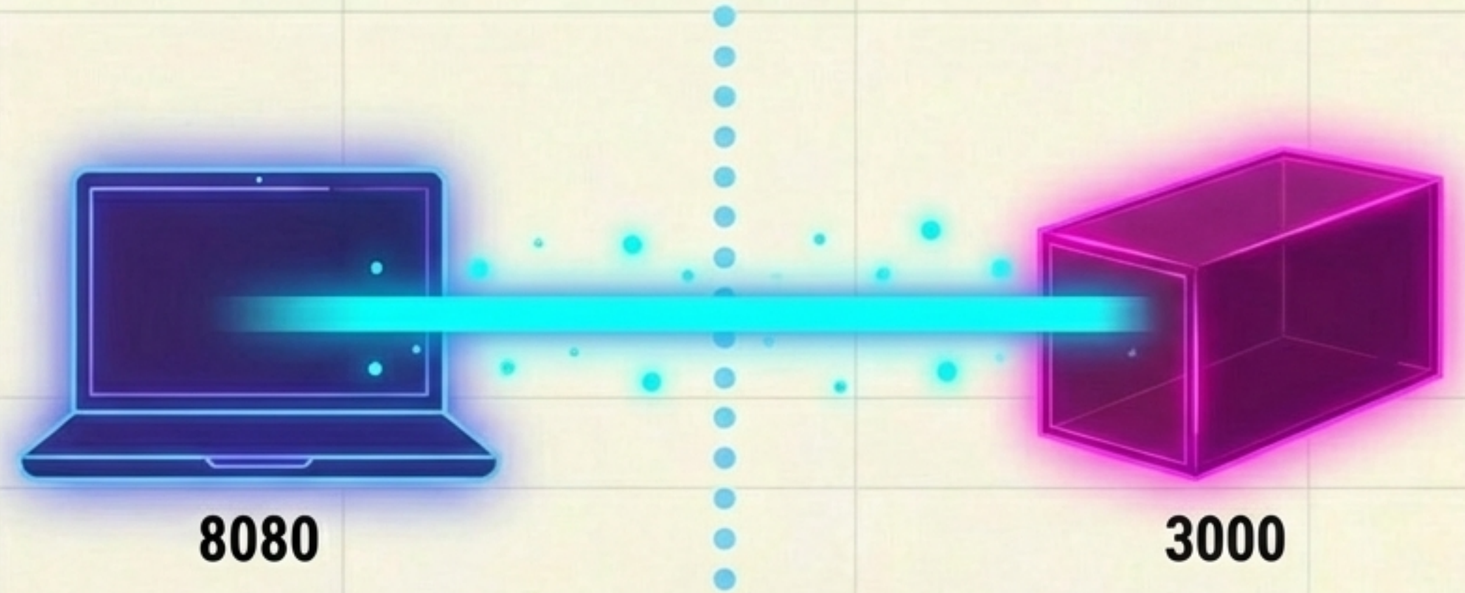
Port Mapping Demystified

Syntax:

`-p hostPort:containerPort`

Example:

```
docker run -p 8080:3000
my-node-app
```



How it Works:

The host laptop listens on 8080. All traffic hitting 8080 is forwarded straight into the isolated container's 3000 port. You visit `localhost:8080` in your browser.

Port Collisions: If 3000 is already taken on your laptop, you can't map to it. Map to 3001:3000 instead. The container's internal port stays whatever the app was programmed to listen on.

The Hazard Map: Common Gotchas

Symptom	Likely Cause	Quick Fix
Container exits instantly	The default CMD process failed.	Check package.json for a missing or crashing start script.
Strange Node module errors	Host node_modules bled into the Linux image.	You forgot to add node_modules to your .dockerignore!
App is unreachable	Traffic isn't crossing from the host.	Did you forget the -p flag? Or did your app bind to 127.0.0.1 instead of 0.0.0.0?

Debugging Triage: If a container fails, run 'docker logs <container-id>'. The Node error stack trace is waiting for you there.

Quick Recap & Cheat Sheet

Syntax Chart

Command	Description
`FROM`	Base image foundation.
`WORKDIR`	Internal execution directory.
`COPY`	Move files host -> image.
`RUN`	Execute command during build.
`EXPOSE`	Document expected internal port.
`CMD`	Default runtime command.
`docker build -t name .`	Construct image.
`docker run -p 8080:3000 name`	Start container.

The Cargo Manifest (Baseline Requirements)

- ✅ **Dockerfile** (The exact recipe)
- ✅ **.dockerignore** (The host shield)
- ✅ **package.json** (Needs a valid start script)
- ✅ **Docker Engine** (Must be running locally)

Active Recall: Container Drills

Tier 1: The Basic Build

Can you write the Docker Bay Six instructions from memory? Can you build ‘node-test-app’ locally and watch the daemon step through the layers?

Tier 2: Traffic Control

Assume your Node app listens on 3000. Can you start the container but intentionally map the host port to 8080? If you ran a second instance, how would you avoid a collision?

Tier 3: Caching Diagnostics

Force a cache invalidation. Write a ‘less helpful’ Dockerfile (COPY .. first). Add a comment to server.js. Rebuild and watch the speed penalty of a busted cache in real-time.

Exam Prep: Solo's Pro-Notes

The DevOps Mindset

We define environments explicitly to eradicate silent drift. Rote memorization isn't the goal; operational consistency is.

The Mental Model

Images = Class (Static Blueprint).
Containers = Object (Live Execution).

The Immutability Rule

If you ever find yourself logging in to a running container to manually fix a configuration file, you have failed the test. Fix the Dockerfile and rebuild the image.

p.s., keep learning!