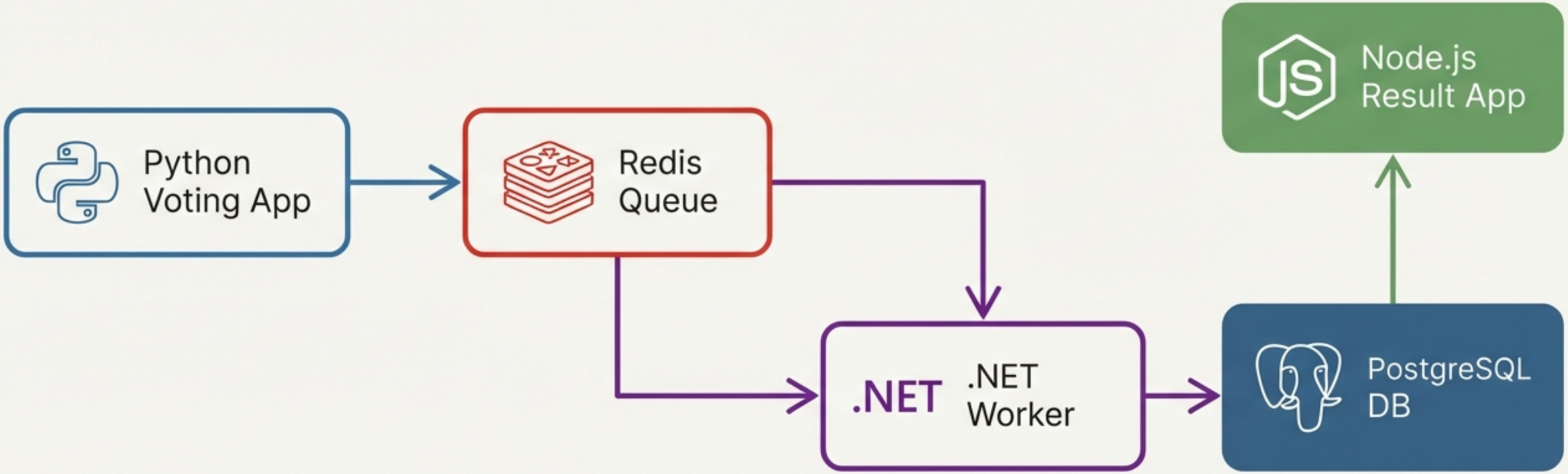
A Docker Journey: From a Single Container to a Full Application Stack

An interactive, self-guided tour through the core concepts of Docker, using a real-world multi-service application as our guide.



The Challenge: Deploying the Multi-Service Voting App

Vote

A Python Flask front-end for users to cast votes.

Redis

An in-memory cache to queue incoming votes.

Worker

A .NET background service to process votes from Redis.

DB

A PostgreSQL database for persistent vote storage.

Result

A Node.js web app to display the aggregated results.

How do we run, connect, configure, and manage these five independent services as a single, cohesive application?

Step 1: Running a Single Service in an Isolated Environment

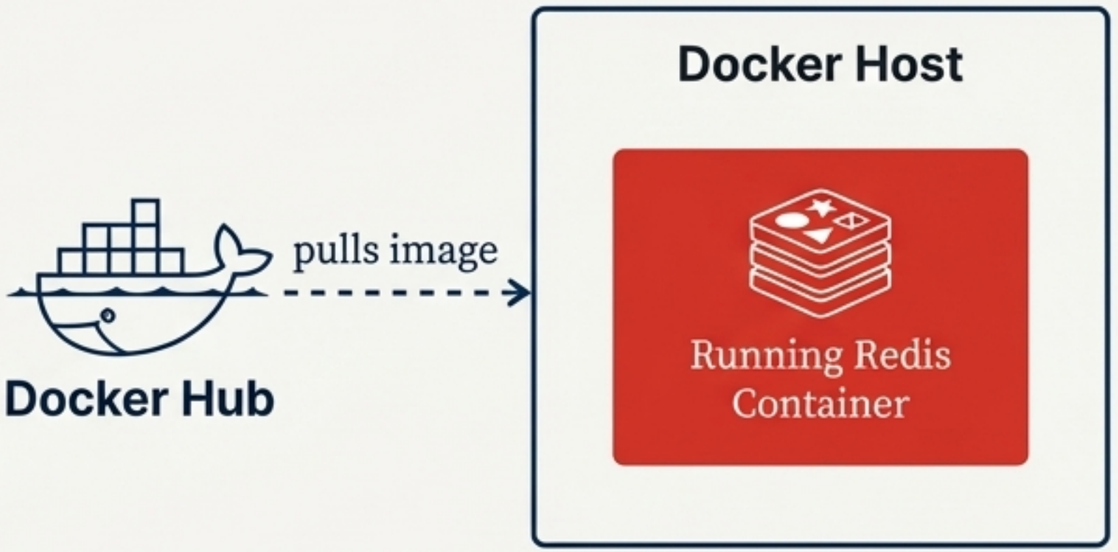
The Problem

Before building our own app, let's run a dependency. How do we get a Redis server running instantly, without a complex installation?

The Solution

The `docker run` command.

```
# Pulls the latest version of Redis from Docker Hub and runs it.
docker run redis

# You can also specify a specific version tag.
docker run redis:4.0
```



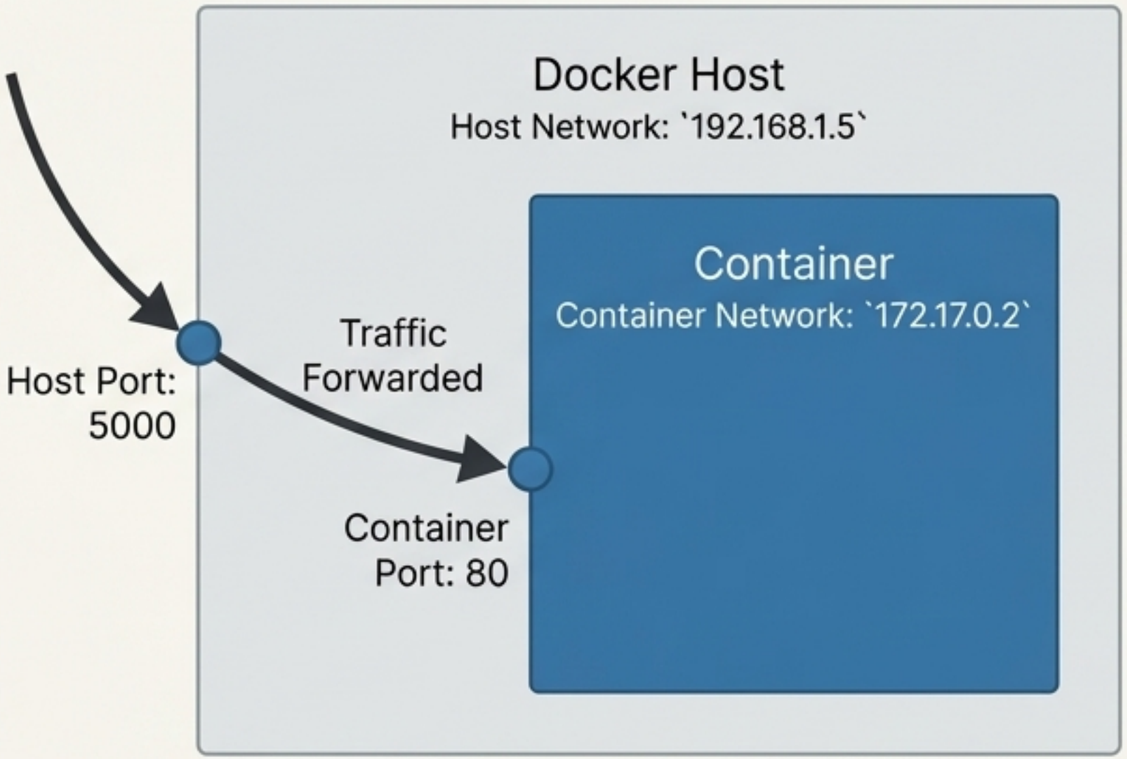
`docker run [image-name]` is the fundamental command to create and start a container from a pre-built image.

Exposing Our Application to the World

Problem: Our containerized web app is running on its own internal network (e.g., at `172.17.0.2:80`), but how do we access it from our browser on the host machine?

Solution: Port Mapping (`-p`). We map a port on the Docker host to a port inside the container.

```
# Maps port 5000 on the host to port 80
inside the container.
docker run -p 5000:80 voting-app
```



Port mapping bridges the gap between the host's network and the container's isolated network.

The Automation Challenge: From a Base OS to a Running App

Manually configuring each service inside a container is repetitive and not scalable. What if we had to set up our Python voting app from scratch?

**Manual Setup Process:

1. **Start with a base Ubuntu OS:**
   ```
   docker run -it ubuntu bash
   ```

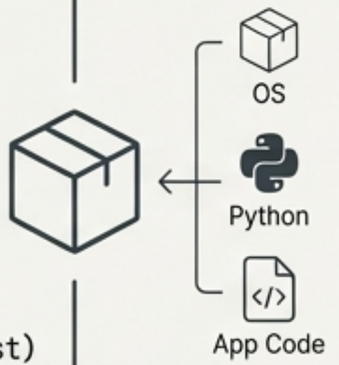
2. **Update package repositories:**
   ```
   apt-get update
   ```

3. **Install OS dependencies:**
   ```
   apt-get install -y python python-pip
   ```

4. **Install Python packages:**
   ```
   pip install flask redis
   ```

5. **Copy application source code into the container:**
   ```
   # (Requires a separate 'docker cp' command from the host)
   COPY app.py /app/
   ```

6. **Set environment variables and run the application:**
   ```
   export FLASK_APP=/app/app.py
   flask run
   ```



Key Insight: This manual process is slow, difficult to reproduce consistently, and hard to version control. There must be a better way.

The Solution: A Repeatable Blueprint with a Dockerfile

A Dockerfile is a simple text file that contains a list of instructions to automatically assemble an image.

Manual Steps

1. Start with a base OS
2. Install dependencies
3. Install Python packages
4. Copy source code
5. Define run command

Dockerfile Blueprint

```
# 1. Start with a base image
FROM python:2.7-alpine

# Set working directory & install packages
WORKDIR /app
ADD requirements.txt /app/
RUN pip install -r requirements.txt  # Handles steps 2 & 3

# 4. Copy application source code
ADD . /app

# 5. Define the command to run on start
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80"]
```

`docker build . -t voting-app`

Key Takeaway: A Dockerfile automates the image creation process, making it repeatable, versionable, and shareable.

The Wall of Text: Managing the Full Stack Manually

We have images for all our services, but launching and connecting them is a **complex, error-prone** mess of commands.

```
docker run -d --name=redis redis
docker run -d --name=db postgres:9.4
docker run -d --name=vote -p 5000:80 --link redis:
redis voting-app
docker run -d --name=result -p 5001:80 --link db
:db result-app
docker run -d --name=worker --link redis:redis --
link db:db worker-app
```

Hard to read and maintain.

Uses deprecated linking feature.

No clear order of startup.

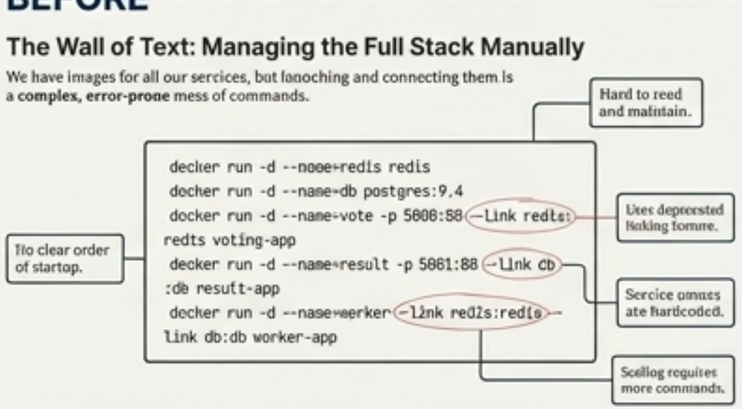
Service names are hardcoded.

Scaling requires more commands.

The Power of Compose: Defining an Entire Application in One File

Docker Compose uses a simple YAML file to define and run multi-container applications.

BEFORE



AFTER

```yaml
version: "3"
services:
  redis:
    image: redis
  db:
    image: postgres:9.4
    environment:
      POSTGRES_PASSWORD: postgres
  vote:
    image: voting-app
    ports:
      - "5000:80"
  worker:
    image: worker-app
  result:
    image: result-app
    ports:
      - "5001:80"
```

`docker-compose up`

Key Takeaway: Docker Compose transforms complex multi-container management into a single, declarative configuration file.

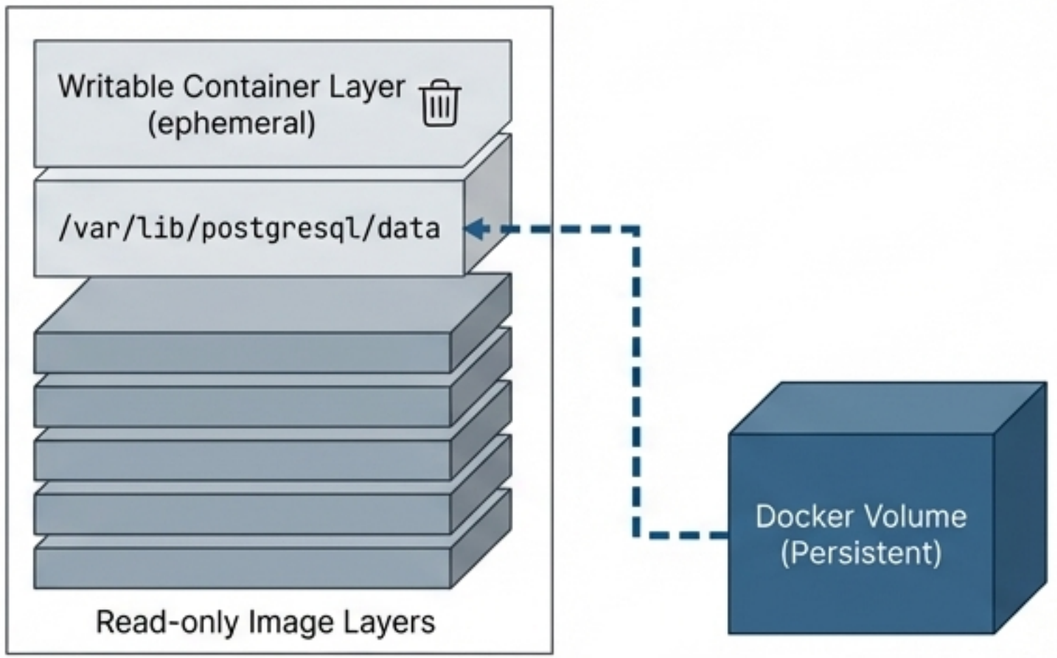
Protecting Your Data:
The Problem of Ephemeral Containers

Problem Statement:
When the `db` container is removed and recreated, all the votes are lost. The container's filesystem is ephemeral.

Solution:
Docker Volumes. Volumes are managed by Docker and exist outside the container's lifecycle, allowing data to persist.

Code Snippet (Docker Compose YAML):

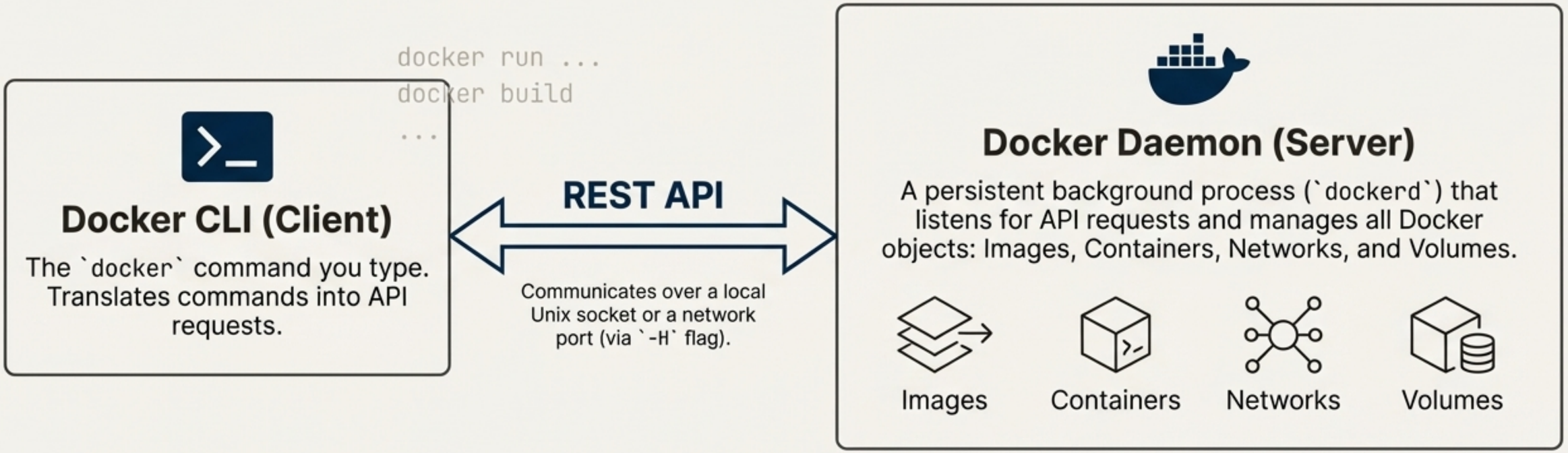
```yaml
services:
  db:
    image: postgres:9.4
    volumes:
      - db-data:/var/lib/postgresql/data

volumes:
  db-data:
```



Key Insight: Volumes decouple data from the container, ensuring that critical information survives container restarts and updates.

Under the Hood: The Docker Engine Architecture

Now that our application is running, let's understand the machinery behind it. Docker is not magic; it's a client-server application.



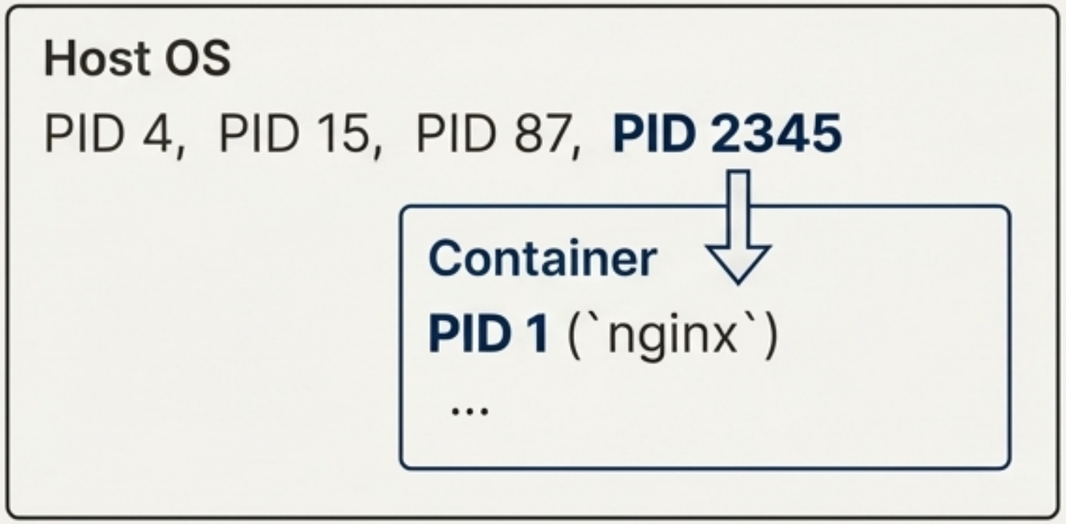
Key Takeaway

The Docker CLI is just a client. The real work is done by the Docker Daemon, which you communicate with via a REST API.

The Pillars of Isolation: Namespaces and Cgroups

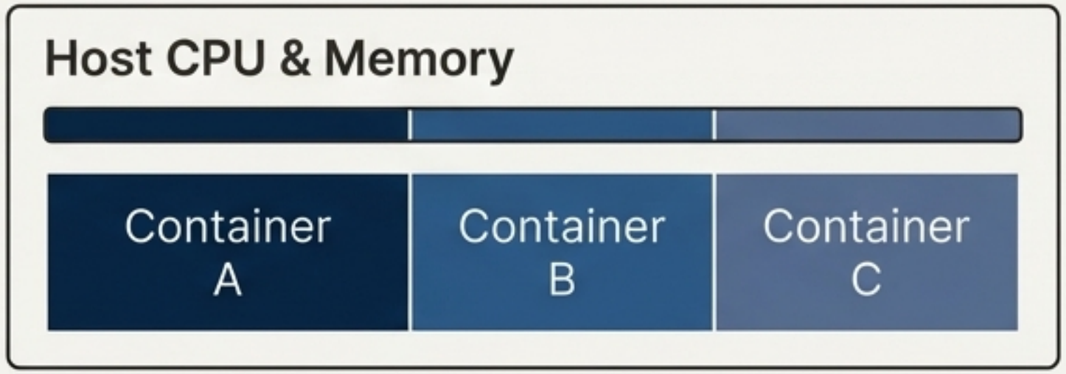
How can a container have its own processes and network, yet share the host's kernel?

Namespaces: A Container's Private View of the World



Namespaces give a container its own isolated view of Process IDs (PIDs), network interfaces, mounts, and more.

Cgroups: Controlling Resource Usage

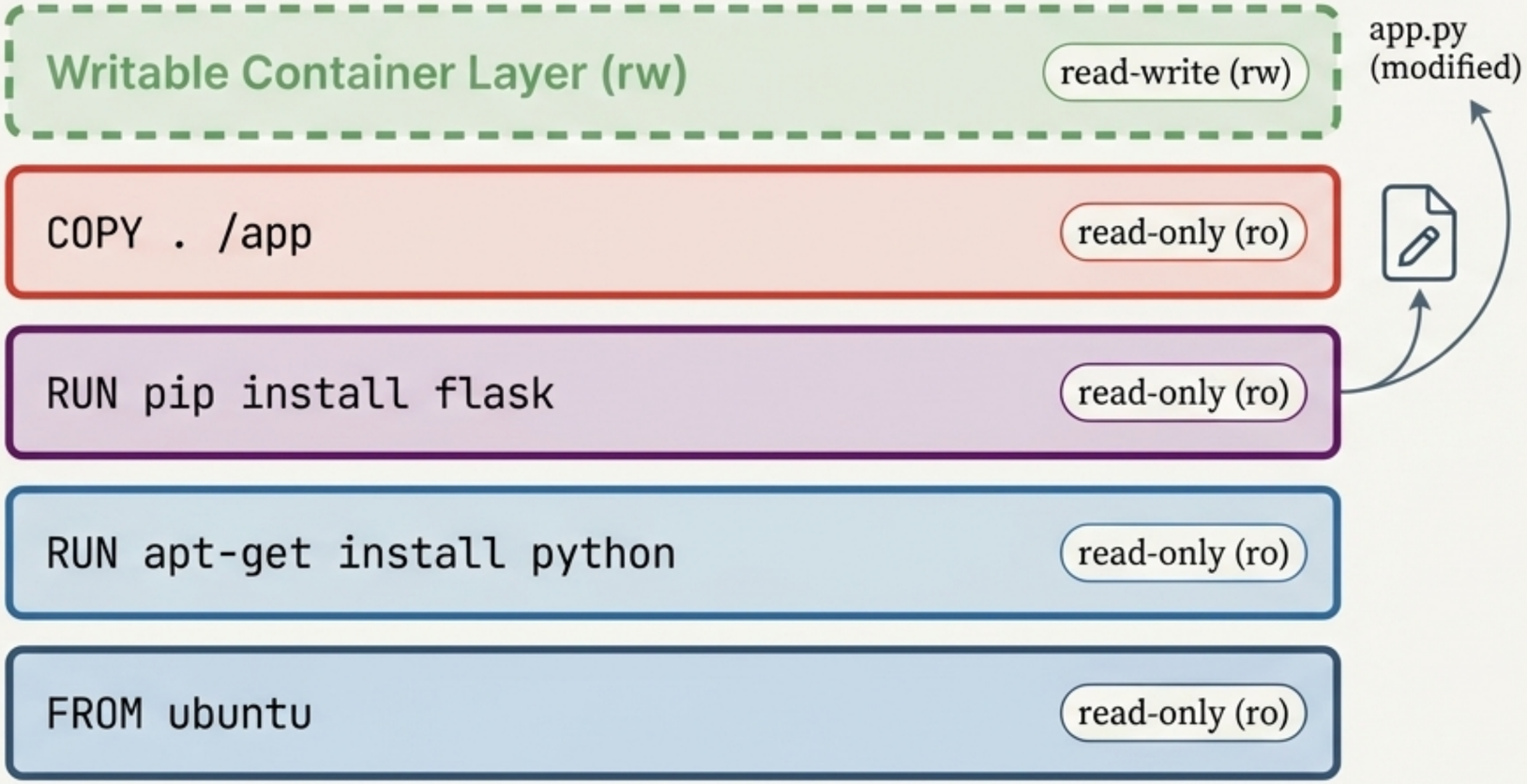


Control Groups (cgroups) limit the amount of CPU, memory, and I/O a container can consume.

```
docker run --cpus=0.5 --memory=512m ...
```

The Anatomy of an Image: A Layered Filesystem

Docker images are not monolithic blobs. Each instruction in a Dockerfile creates a read-only layer. This makes images efficient to build, store, and share.



Copy-on-Write Explained

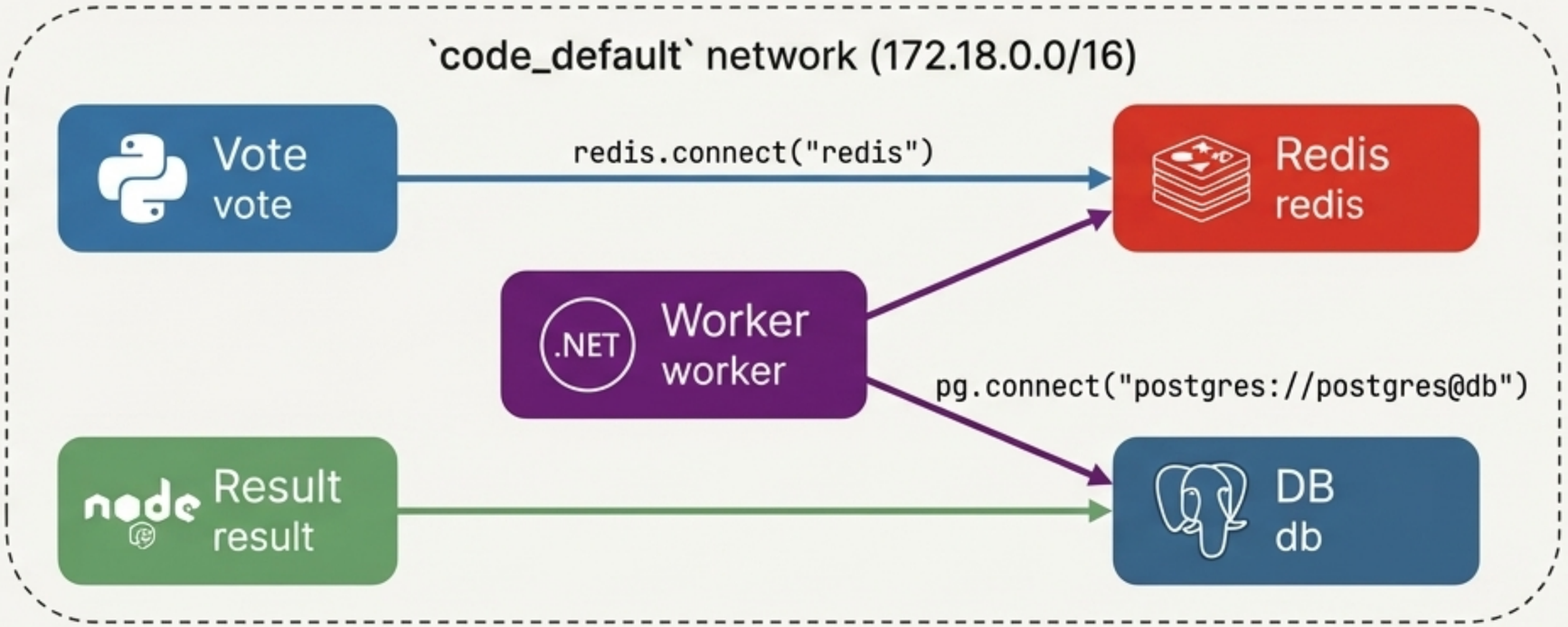
When you modify a file (e.g., *app.py*) that exists in a lower layer, Docker doesn't change the original. original. Instead, it copies the file up to the writable layer and applies the change there. The original image remains untouched.

This is efficient and enables layer sharing between multiple images and containers.

Connecting the Pieces: How Services Talk to Each Other

How does the `vote` container know how to find `redis`? How does `worker` find `db`?

Docker Compose automatically creates a private **bridge network** for all services defined in the file.



Built-in DNS: Docker provides a DNS server on the network. Each container can resolve the hostname of any other container on the same network. For example, the name ‘redis’ resolves to the IP address of the Redis container. This removes the need for deprecated `--link` flags or hardcoded IP addresses.

Storing and Sharing Images: The Role of Registry of the Figsy

A registry is a storage system for Docker images. It's where you `pull' base images from and `push' your own custom images to.

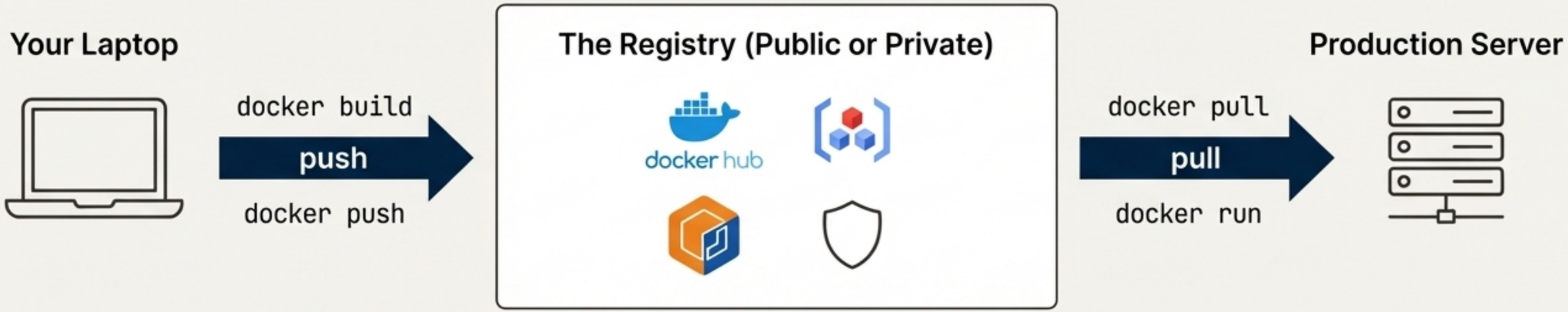


Image Naming Convention

`[registry_host]/[username_or_org]/[repository]:[tag]`

- `docker.io/library/redis:latest` (default)
- `gcr.io/kubernetes-e2e-test-images/agnhost:2.26`
- `private-registry.io/my-company/voting-app:1.2`

Key Commands: `docker login`, `docker push`, `docker pull`

Build Once, Run Anywhere: Docker's Cross-Platform Promise

The entire workflow we've just explored—from `docker run` to Docker Compose—provides a standardized environment that works consistently across different development and production machines. Docker abstracts away the differences in the underlying host OS.

Linux

Native. Runs containers directly on the host kernel.

Windows

Runs Linux containers via a lightweight VM using Hyper-V / WSL2.

macOS

Runs Linux containers via a lightweight VM using the HyperKit virtualization framework.

By containerizing your application with Docker, you create a portable, self-contained package that eliminates the classic "it works on my machine" problem, streamlining development, testing, and deployment.