DEVOPS IN ACTION WORKSHOP

# Day1

Why Docker?
Why Docker Compose?

data infographics

สร้างใช้เอง = In-House

ผู้ว่าจ้าง = CONTRACTEE
ผู้รับจ้าง = CONTRACTOR

ผู้ว่าจ้าง Client
ผู้รับจ้าง Contractor

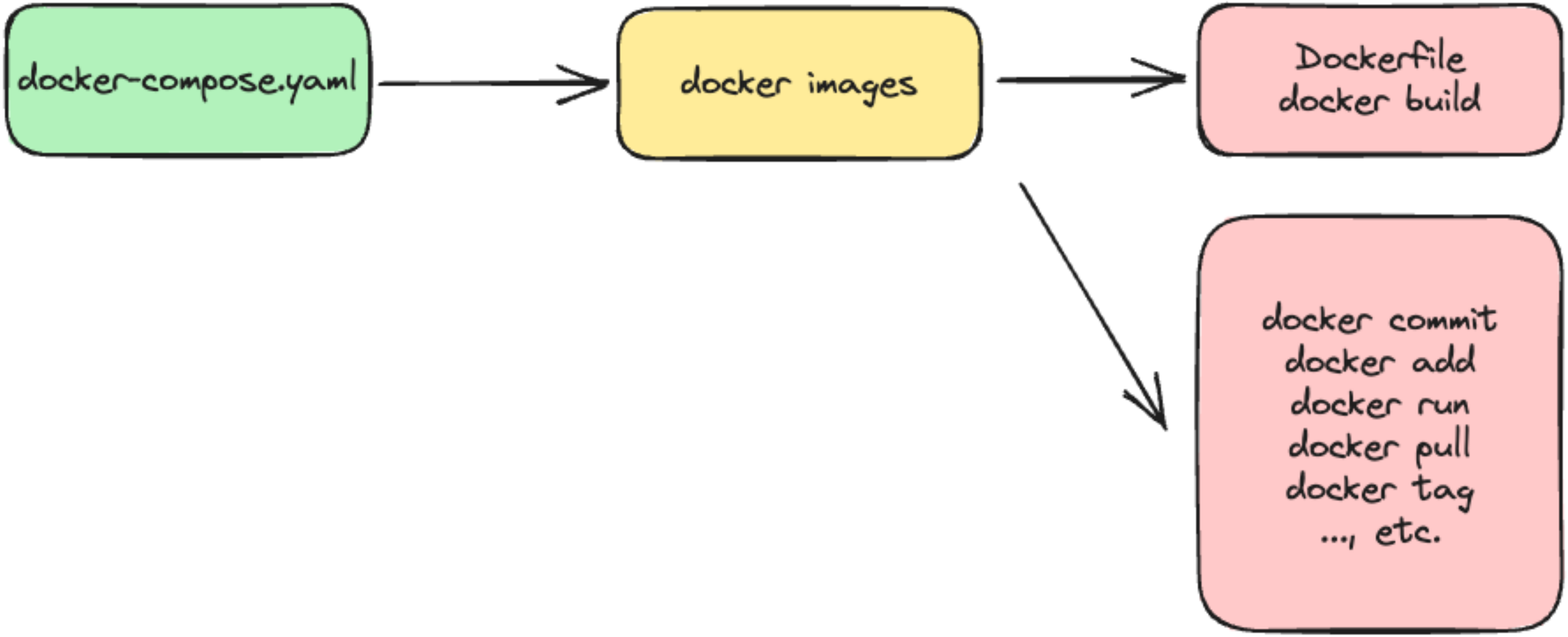
1. Green Field
2. Legacy

1. Cloud
2. On Premise
3. Mixed

Cost & Benefit

ผู้ว่าจ้าง Principal
ผู้รับจ้าง (ชั้นต้น) Contractor
ผู้รับจ้าง (ต่อ) Subcontractor

data infographics



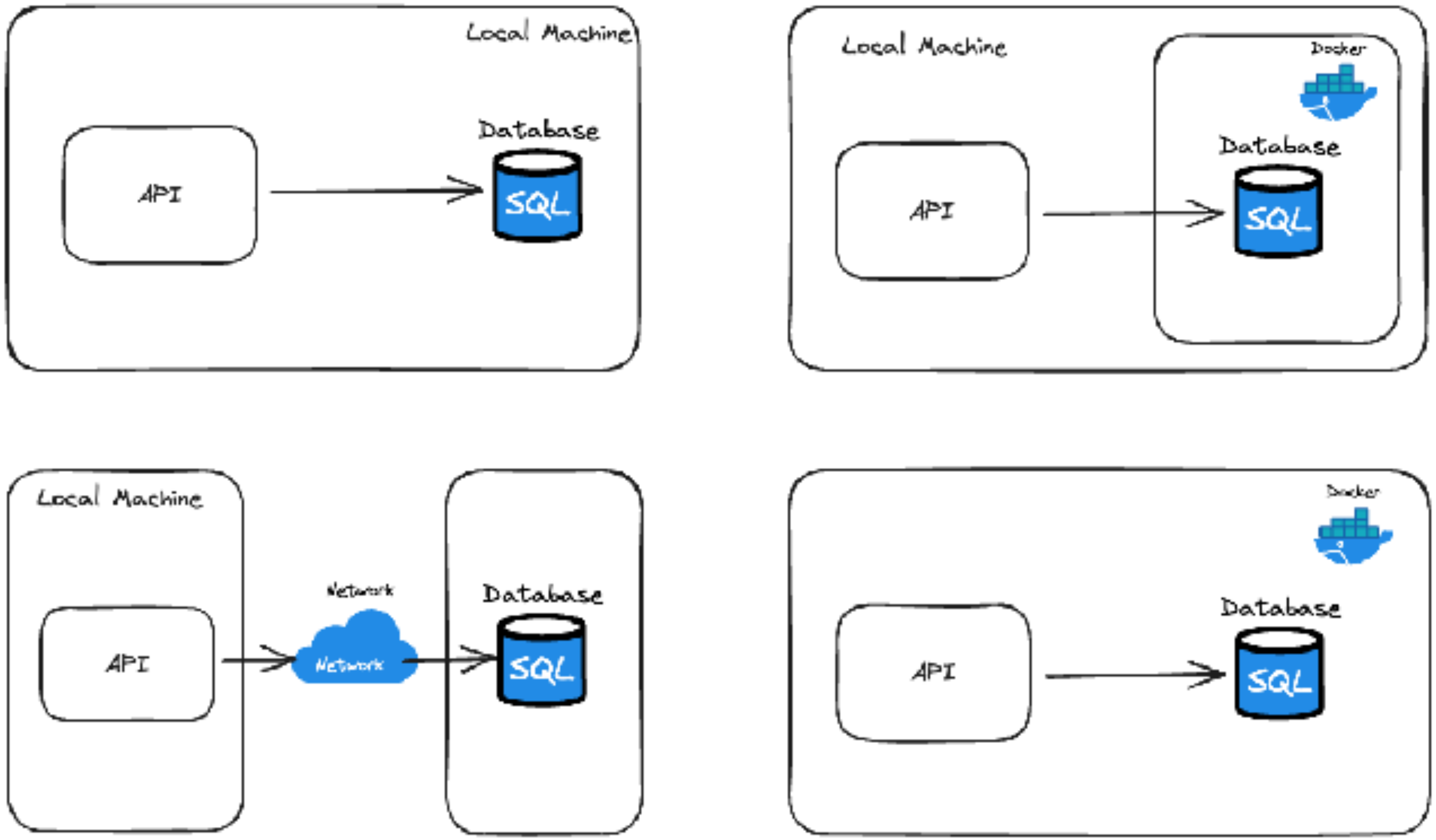
docker-compose.yaml

docker images

Dockerfile
docker build

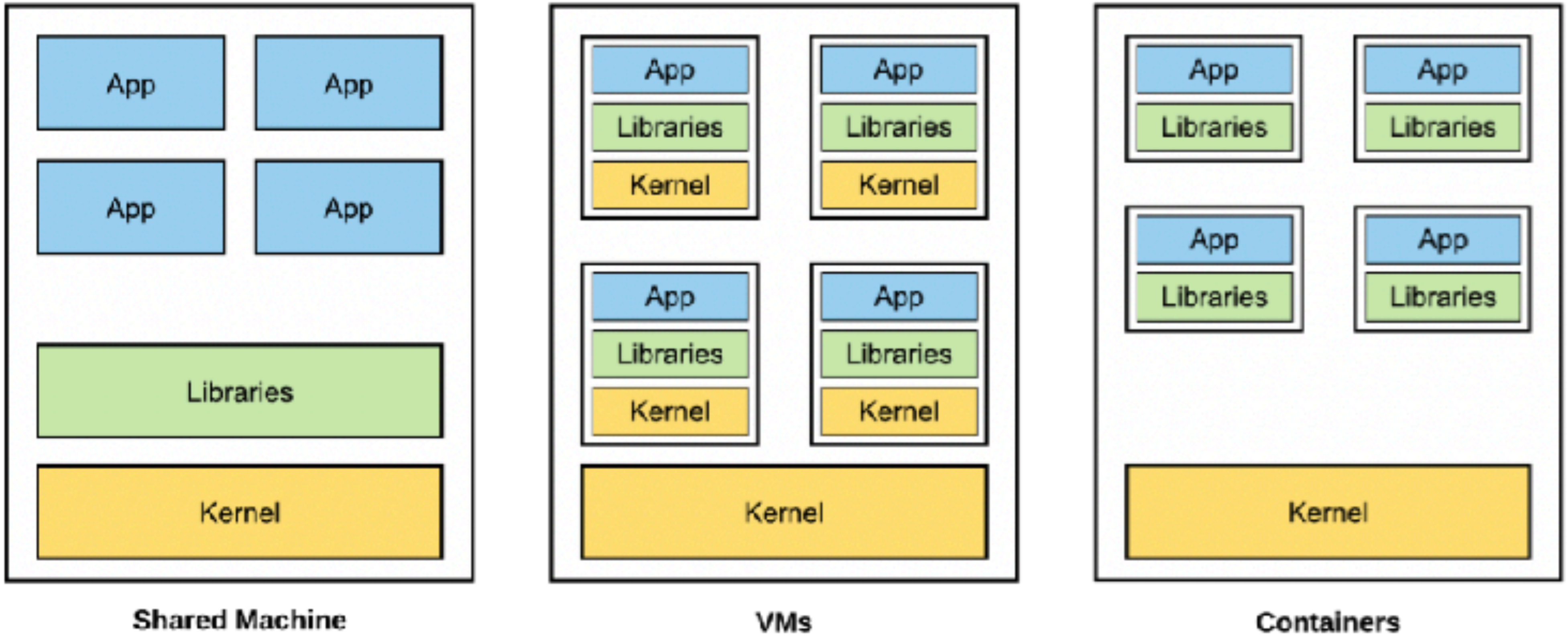
docker commit
docker add
docker run
docker pull
docker tag
..., etc.

data infographics



Docker History

Evolution of Containers



from: Getting Started with Kubernetes

Several Specific Benefits of Containers

- Language Flexibility
- Isolation Without Overhead: light weight
- Developer Efficiency: Isolating Dependencies(libs, configuration)
- **Reproducibility**: Containers make it easier to reproduce your application environment.

The 12 Factor App & Container Principle

The 12 Factor App

1. Codebase
   One codebase tracked in revision control, many deploys
2. Dependencies
   Explicitly declare and isolate dependencies
3. Config
   Store config in the environment
4. Backing services
   Treat backing services as attached resources
5. Build, release, run
   Strictly separate build and run stages
6. Processes
   Execute the app as one or more stateless processes
7. Port binding
   Export services via port binding
8. Concurrency
   Scale out via the process model
9. Disposability
   Maximize robustness with fast startup and graceful shutdown
10. Dev/prod parity
    Keep development, staging, and production as similar as possible
11. Logs
    Treat logs as event streams
12. Admin processes
    Run admin/management tasks as one-off processes

Principle of Container-based Application Design

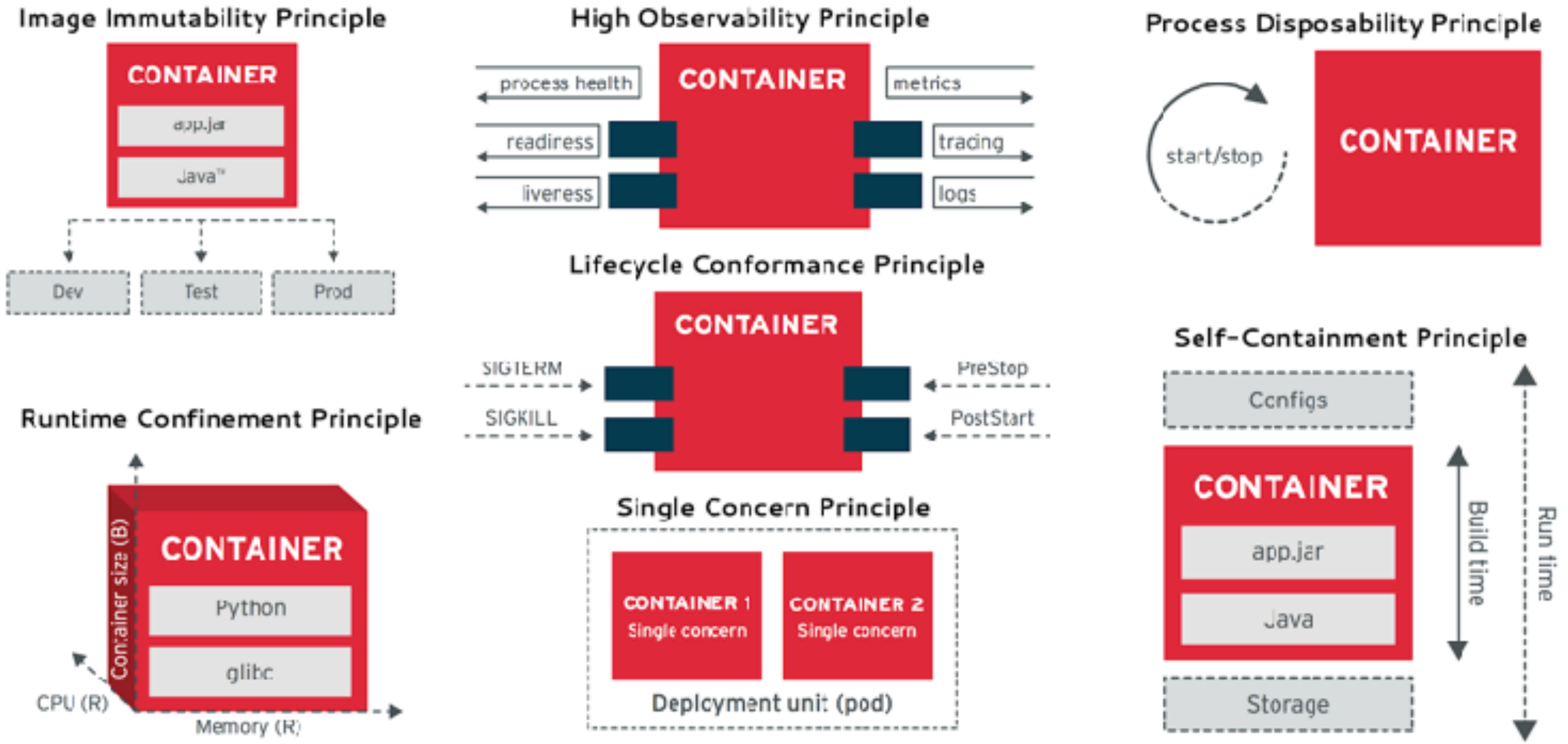


Image Immutability Principle

High Observability Principle

Process Disposability Principle

Lifecycle Conformance Principle

Runtime Confinement Principle

Single Concern Principle

Self-Containment Principle

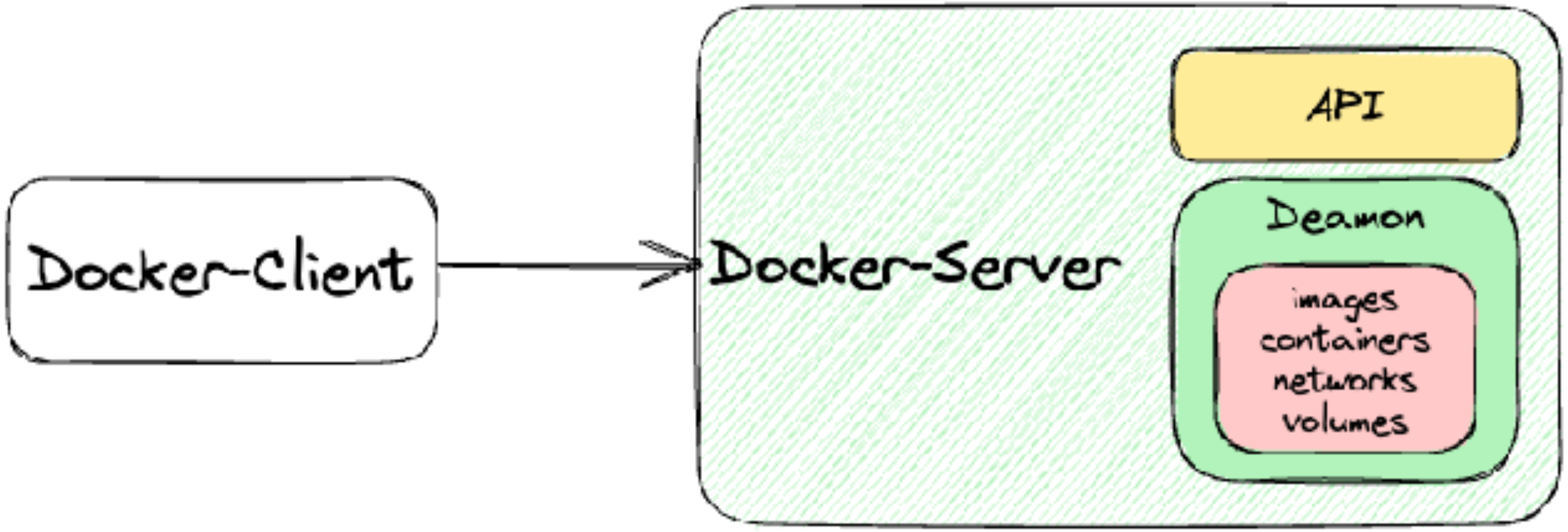
Hello, World

Docker run

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.(amd64)
3. The Docker daemon created a new container from that image which runs the executable that
produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.
...

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

Docker Client & Server

$ **docker version**



Docker Command

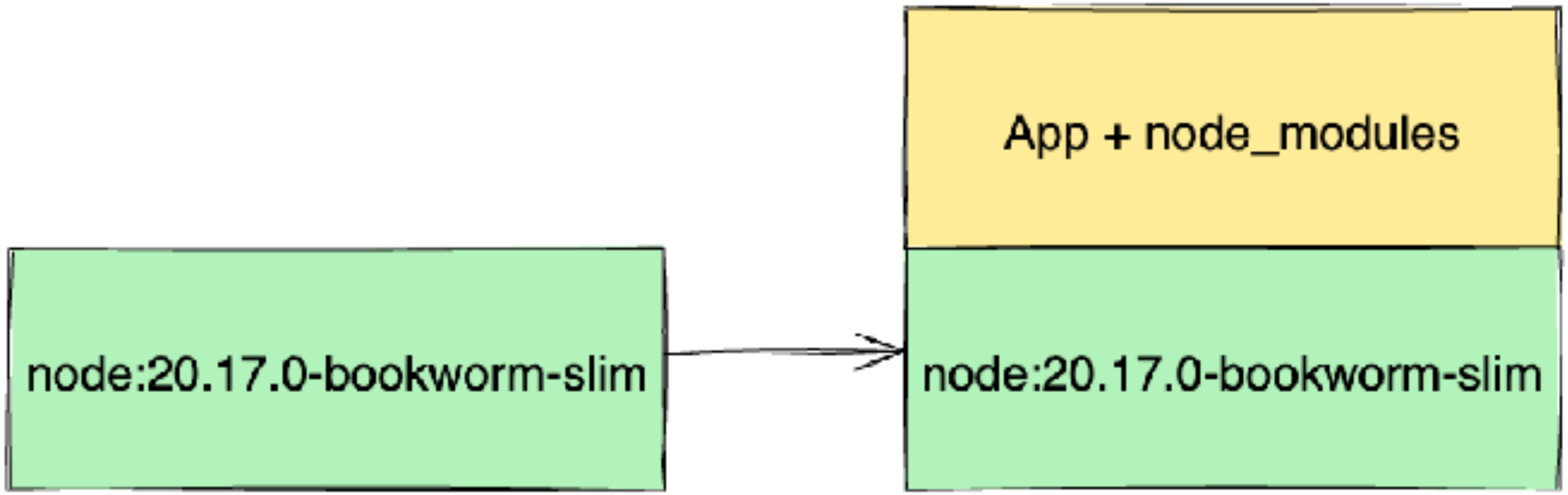
```
$ docker

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Common Commands:
  run         Create and run a new container from an image
  exec        Execute a command in a running container
  ps          List containers
  build       Build an image from a Dockerfile
  pull        Download an image from a registry
  push        Upload an image to a registry
  images      List images
  login       Log in to a registry
  logout      Log out from a registry
  search      Search Docker Hub for images
  version     Show the Docker version information
  info        Display system-wide information
```

Docker 101

Scenario

docker pull
docker container run

docker cp
docker commit



node:20.17.0-bookworm-slim

App + node_modules

node:20.17.0-bookworm-slim

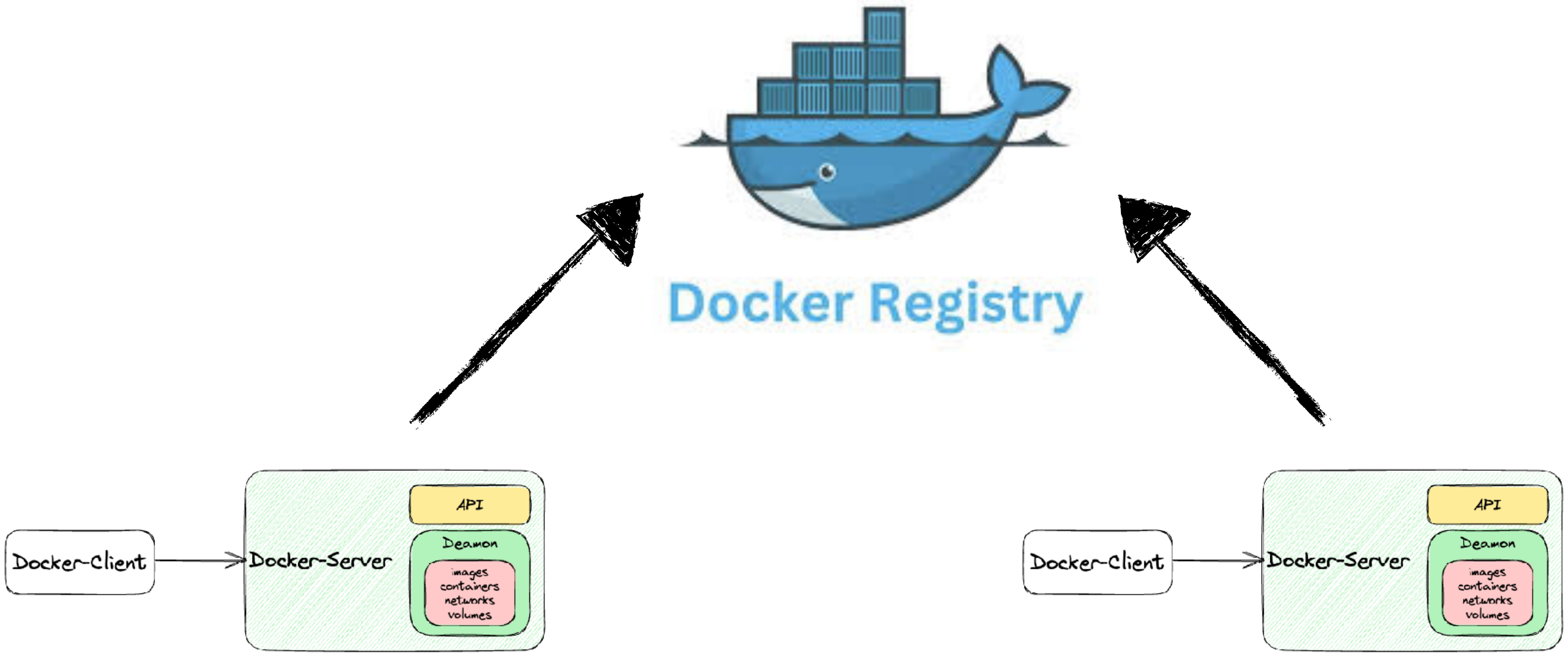
contianer name: node

image: hello:0.0.1

Create docker image from scratch

```
install_packages:
    cd src && npm install

run_app:
    cd src && npm start

pull_based_image:
    docker pull node:20.17.0-bookworm-slim

run_based_image:
    docker container run --name node node:20.17.0-bookworm-slim

copy_src:
    docker container cp ./src node:/root/

commit_change:
    docker container commit node hello:0.0.1

run_hello:
    docker container run -p 3000:3000 --name hello-api hello:0.0.1 node /root/src/index.js

rm_hello:
    docker container rm -f hello-api
```

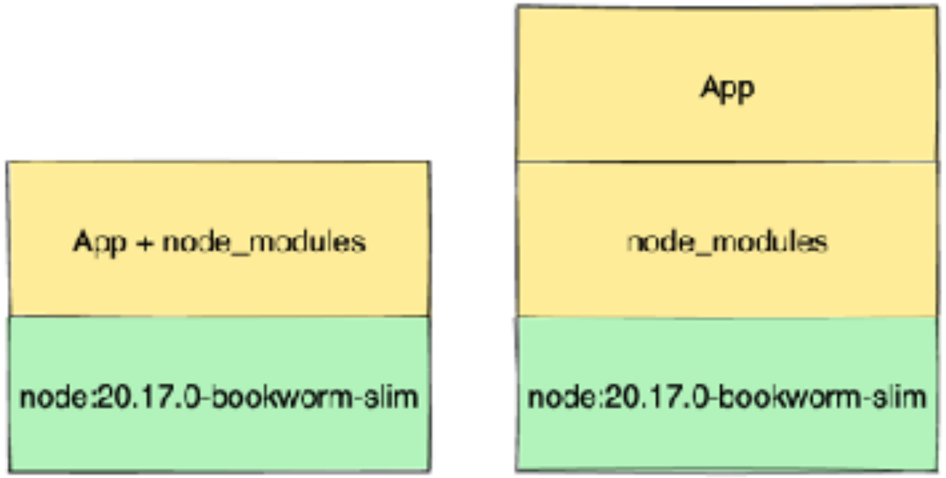
Store and Share Docker Images



- https://hub.docker.com/_/dockerhub_username: `official images`
- https://hub.docker.com/u/dockerhub_username: `user profiles`
- https://hub.docker.com/r/dockerhub_username: `repositories`

Dockerfile

Scenario



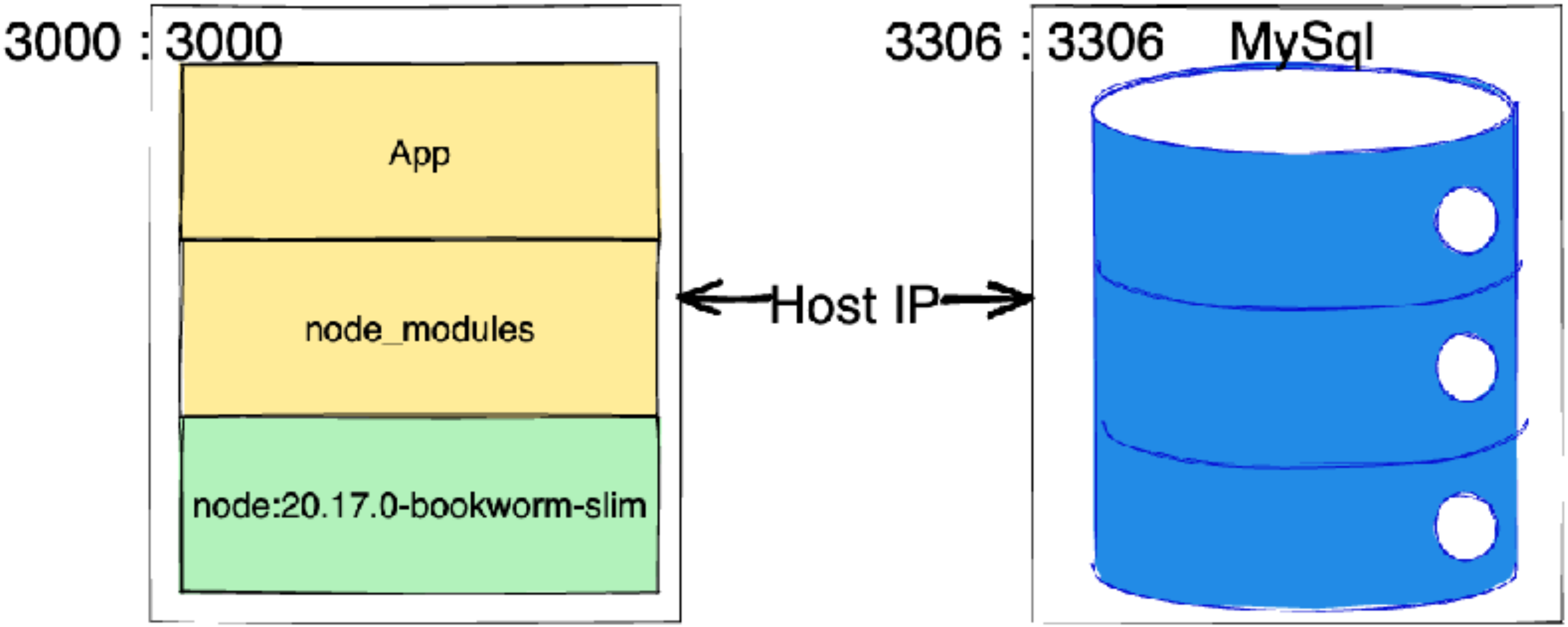
```
# -------------- The build image --------------
FROM node:20.17.0-bookworm AS build
WORKDIR /usr/src/app
COPY package*.json /usr/src/app/
RUN npm ci --only=production

# -------------- The production image ---------
FROM node:20.17.0-bookworm-slim
ENV NODE_ENV=production
USER node
WORKDIR /usr/src/app
COPY --chown=node:node --from=build /usr/src/app/
node_modules /usr/src/app/node_modules
COPY --chown=node:node . /usr/src/app
EXPOSE 3000
CMD ["node", "index.js"]
```

Create docker image from Dockerfile

```
build_hello:
    cd src && docker image build -t hello:0.0.2 .

run_hello:
    docker container run -e "PORT=3000" -p 3000:3000 --name hello-api hello:0.0.2

rm_hello:
    docker container rm -f hello-api

# --------- dump-init----------

build_hello_dump_init:
    cd src && docker image build -f Dockerfile-dump-init -t hello:0.0.3 .

run_hello_dump_init:
    docker container run -e "PORT=3000" -p 3000:3000 --name hello-api hello:0.0.3
```

Working with Database

Scenario


3000 : 3000
App
node_modules
node:20.17.0-bookworm-slim
Host IP
3306 : 3306
MySql

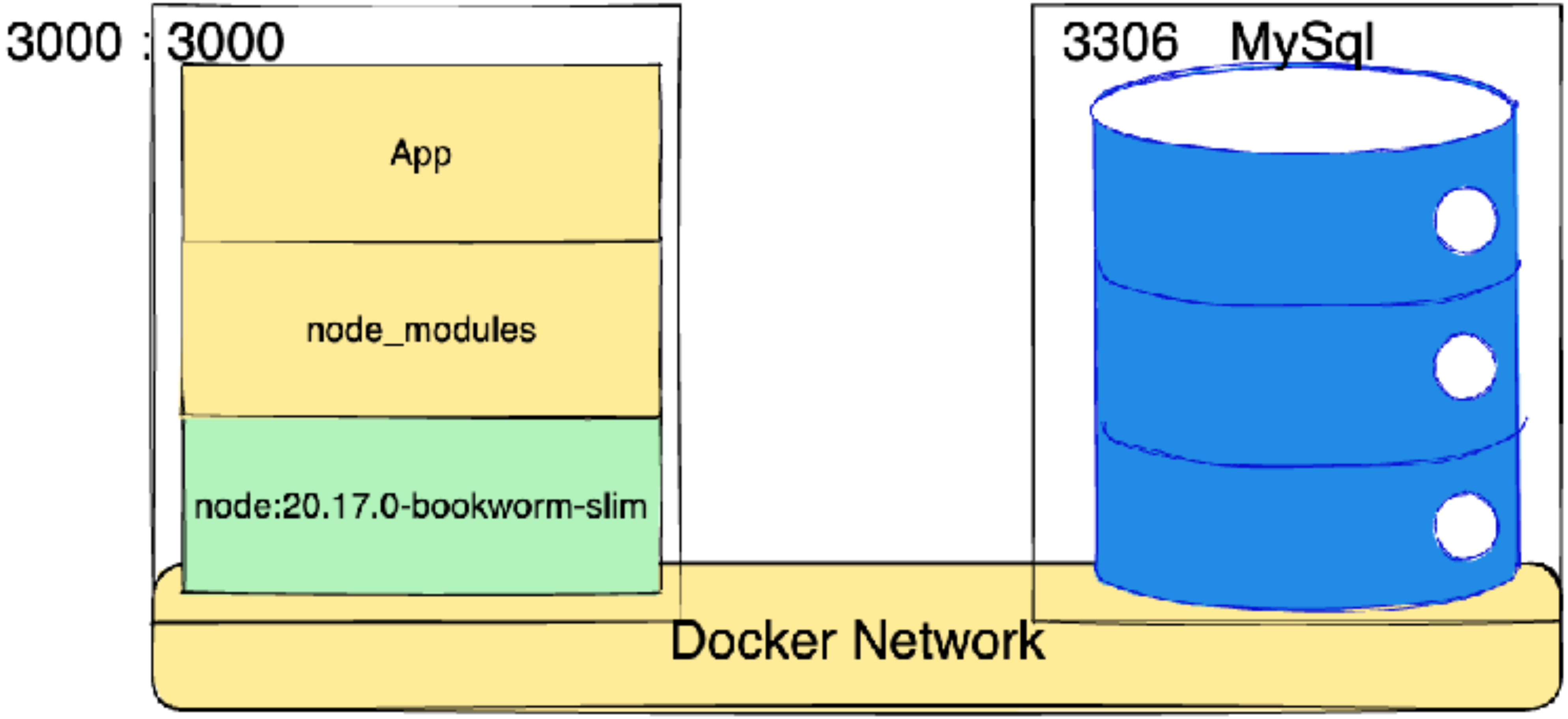
Create API and Call Database via Host IP

```
build_api:
    cd src && docker image build -t api:0.0.1 .

start_api:
    docker container run --rm -d --name api \
    -e DB_HOST=<IP> \
    -e DB_USER=admin \
    -e DB_PASSWORD=password \
    -e DB_NAME=mydatabase \
    -e DB_PORT=3306 \
    -e PORT=3000 \
    -p 3000:3000 \
    api:0.0.1
```

```
start_db:
    docker container run --rm --name mysql9 \
    -v ./data/:/docker-entrypoint-initdb.d/ \
    -e MYSQL_ROOT_PASSWORD=password \
    -e MYSQL_DATABASE=mydatabase \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=password \
    -p 3306:3306 \
    -d mysql:9.0.1-oraclelinux9
```

Docker Network

Scenario


3000 : 3000
App
node_modules
node:20.17.0-bookworm-slim
3306 MySql
Docker Network

Communication via Docker Network

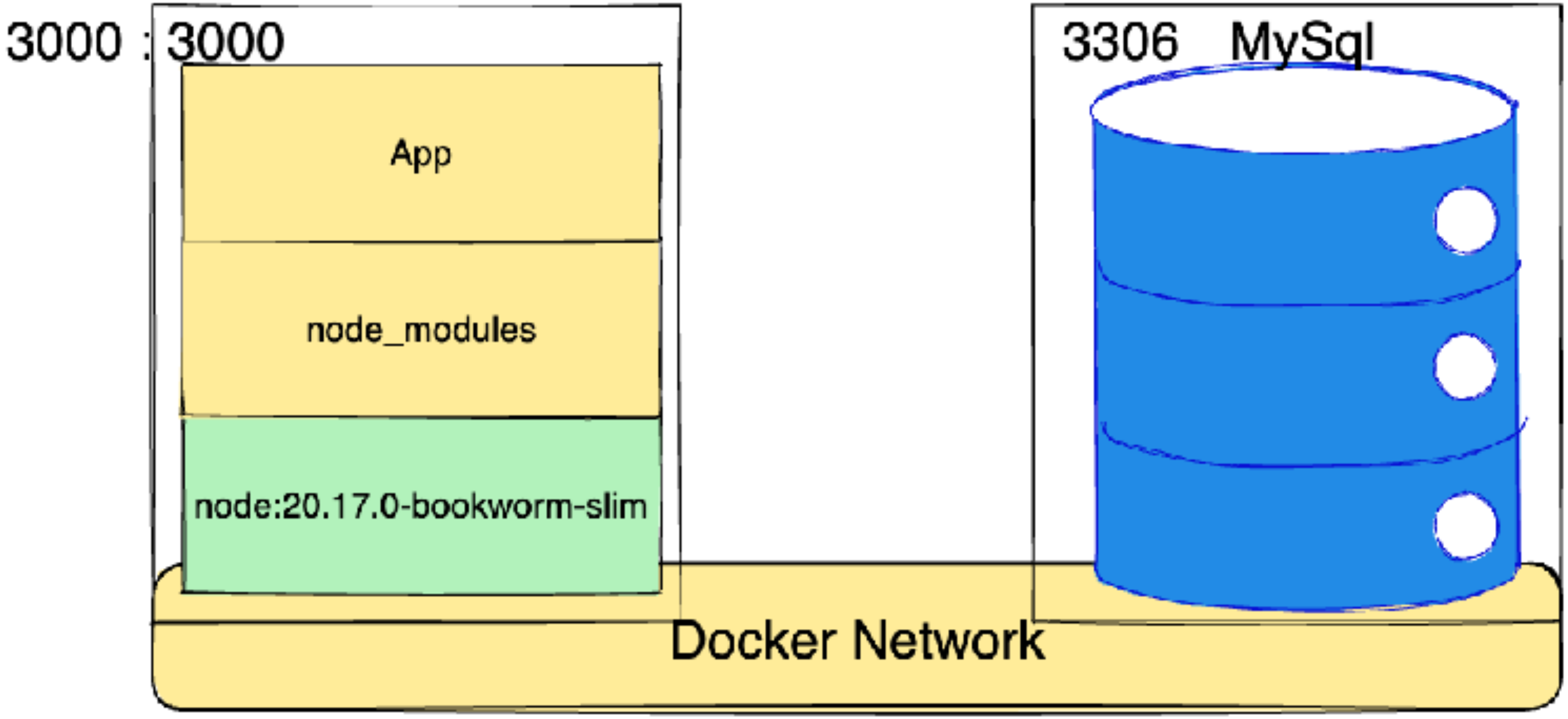
```
start_api_with_network:
    docker container run --rm -d --name api \
    -e DB_HOST=mysql9 \
    -e DB_USER=admin \
    -e DB_PASSWORD=password \
    -e DB_NAME=mydatabase \
    -e DB_PORT=3306 \
    -e PORT=3000 \
    -p 3000:3000 \
    --network hello \
    api:0.0.1
```

```
start_db_with_network:
    docker container run --rm --name mysql9 \
    -v ./data/:/docker-entrypoint-initdb.d/ \
    -e MYSQL_ROOT_PASSWORD=password \
    -e MYSQL_DATABASE=mydatabase \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=password \
    --network hello \
    -d mysql:9.0.1-oraclelinux9
```

```
create_hello_network:
    docker network create hello
list_network:
    docker network ls
inspect_network:
    docker network inspect hello
delete_hello_network:
    docker network rm hello
```

Docker Compose

Scenario


3000 : 3000
App
node_modules
node:20.17.0-bookworm-slim
3306 MySql
Docker Network

YAML

```
api:
  image: api:0.0.1
  container_name: api
  # restart: always
  ports:
    - 3000:3000
  networks:
    - hello
  depends_on:
    db:
      condition: service_healthy
  environment:
    - DB_HOST=db
    - DB_USER=admin
    - DB_PASSWORD=password
    - DB_NAME=mydatabase
    - DB_PORT=3306
    - PORT=3000
```

```
{
  "api": {
    "image": "api:0.0.1",
    "container_name": "api",
    "ports": [ "3000:3000" ],
    "networks": [ "hello" ],
    "depends_on": {
      "db": { "condition": "service_healthy" }
    },
    "environment": [
      "DB_HOST=db",
      "DB_USER=admin",
      "DB_PASSWORD=password",
      "DB_NAME=mydatabase",
      "DB_PORT=3306",
      "PORT=3000"
    ]
  }
}
```

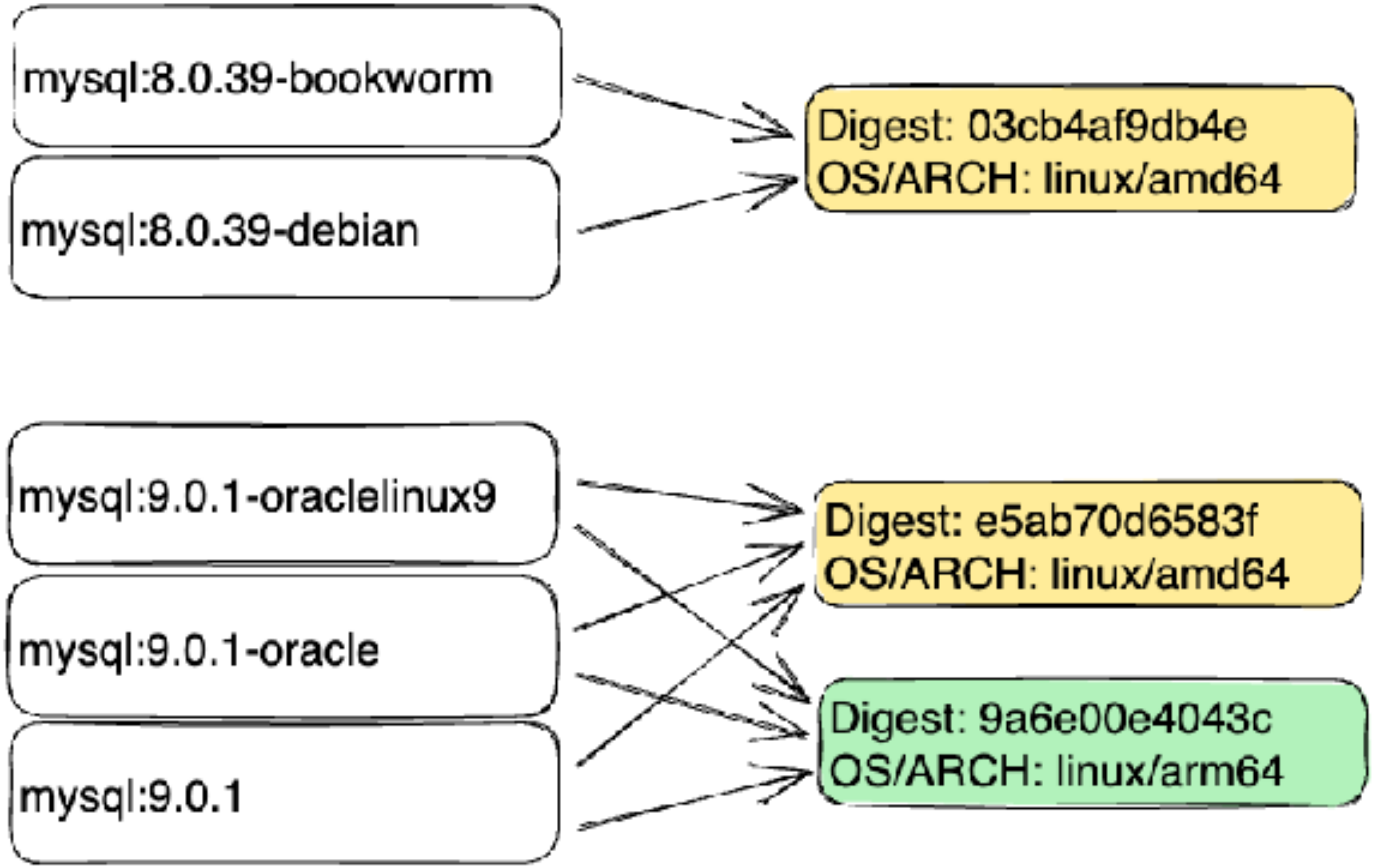
Imperative vs Declarative

```
create_hello_network:
    docker network create hello

build_api:
    cd src && docker image build -t api:0.0.1 .

start_api_with_network:
    docker container run --rm -d --name api \
    -e ... \
    -p 3000:3000 \
    --network hello \
    api:0.0.1

stop_api:
    docker container stop api

delete_hello_network:
    docker network rm hello
```

```
api:
  image: api:0.0.1
  build:
    context: src
    dockerfile: Dockerfile
  container_name: api
  ports:
    - 3000:3000
  networks:
    - hello
  depends_on:
    db:
      condition: service_healthy
  environment:
    - DB_HOST=db
    - DB_USER=admin
    - DB_PASSWORD=password
    - DB_NAME=mydatabase
    - DB_PORT=3306
    - PORT=3000
```

Docker Tags

data infographics



mysql:8.0.39-bookworm

mysql:8.0.39-debian

Digest: 03cb4af9db4e
OS/ARCH: linux/amd64

mysql:9.0.1-oraclelinux9

mysql:9.0.1-oracle

mysql:9.0.1

Digest: e5ab70d6583f
OS/ARCH: linux/amd64

Digest: 9a6e00e4043c
OS/ARCH: linux/arm64

data infographics

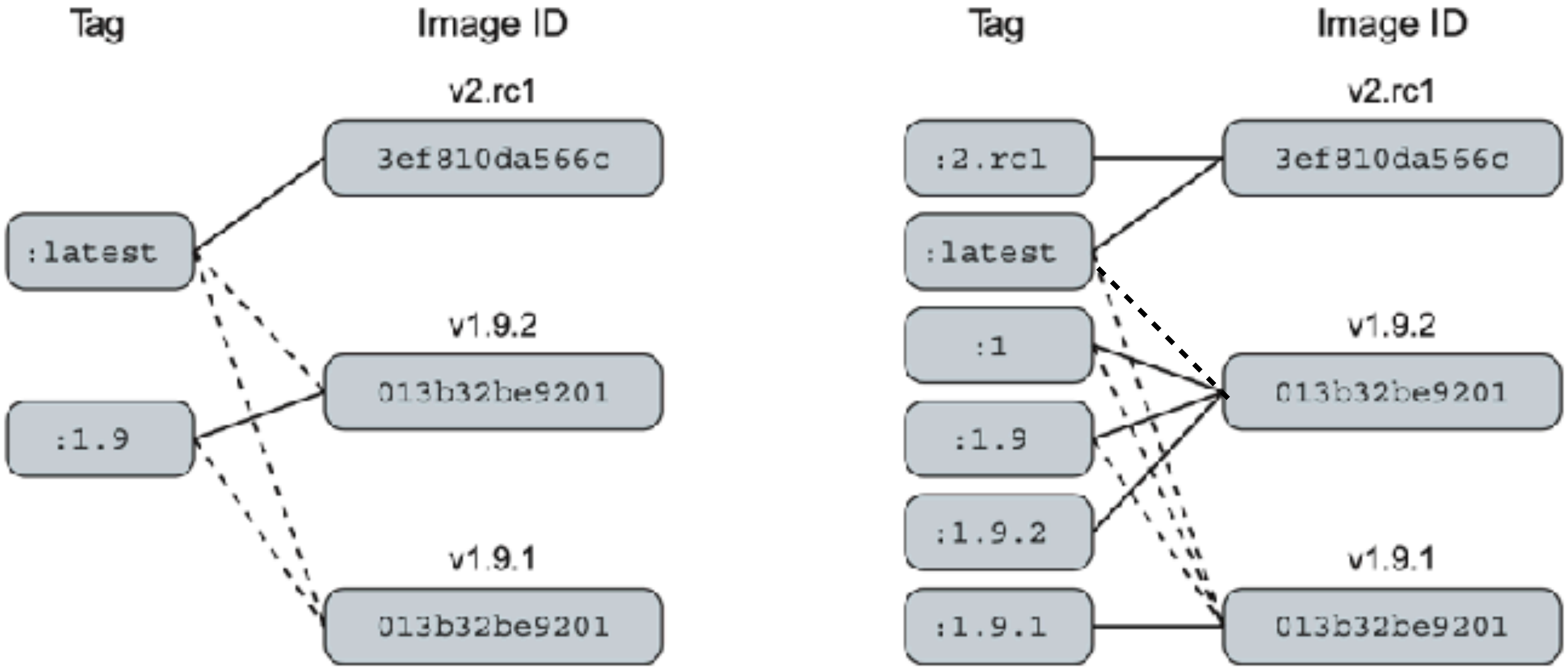


Figure 7.7 Two tagging schemes (left and right) for the same repository with three images. Dotted lines represent old relationships between a tag and an image.

from: Docker in Action, 2 edition

Docker Registry

data infographics

https://distribution.github.io/distribution/

DISTRIBUTION

```
docker pull registry:2.8.3
docker run -d -p 5000:5000 --restart always --name registry registry:2.8.3

docker tag api:0.0.1 localhost:5000/api:0.0.1
docker push localhost:5000/api:0.0.1
docker image rm api:0.0.1 localhost:5000/api:0.0.1
docker pull localhost:5000/api:0.0.1
```

Security for Docker 101

Static Scan with Trivy

https://aquasecurity.github.io/trivy

```
trivy image <image>
trivy image api:0.0.1

trivy image --format template --template "@contrib/sarif.tpl" -o report.sarif <image>
trivy image --format template --template "@contrib/sarif.tpl" -o report.sarif api:0.0.1
```

thanks