Docker

FIUBA - 75.10 Técnicas de Diseño


IT WORKS ON MY MACHINE
THEN WE'LL SHIP YOUR MACHINE
AND THAT IS HOW DOCKER WAS BORN

Tradicionalmente...

Cómo configurar una aplicación en un servidor:

- Acceder directamente al sistema a configurar
- Usar herramientas interactivas para inspeccionar y modificar el sistema

Problemas

- Altamente repetitivo
- Gran probabilidad de error
- Interacciones entre aplicaciones
- Es difícil reconstruir la configuración de un servidor

Máquinas Virtuales

Emulación de una máquina real

- Provee aislamiento total
- Única opción para ejecutar programas de otras arquitecturas de hardware

Algunas herramientas:

- VirtualBox
- VMWare
- Vagrant

Infrastructure as Code (IaC)

En lugar de realizar modificaciones manualmente, se usa un lenguaje de programación específico para el dominio de configuraciones de máquinas

Algunas herramientas:

- Chef
- Puppet
- Ansible
- Terraform

Infraestructura Declarativa

Las herramientas de Infraestructura como Código suelen ser declarativas

Declarativo: Especifica **qué** objetivo lograr
Imperativo: Especifica **cómo** lograr el objetivo

Permite abstraernos del estado actual

```
- hosts: webservers
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf
```

Ejemplo Ansible

Mascotas vs Ganado

Infraestructura Mascota

- Nombres específicos
- Unicas
- Cuidadas

- Mantenimiento delicado
- Presentes aunque no sean necesarios

- Servicios legacy
- Servicios unicos

Infraestructura Ganado

- Numero de serie
- Practicamente identicas
- Reemplazable

- "Mantenimiento" creando otra instancia
- Se crean y destruyen a necesidad

- Servicios sin estado interno relevante

Linux Namespaces

Separa recursos del sistema, para que diferentes grupos de procesos tengan acceso a diferentes conjuntos de recursos.

Algunos namespaces interesantes:

- mnt (Sistema de archivos)
- pid (Listado de procesos)
- net (Interfaces de red)
- ipc (Comunicación entre procesos)

Suelen ser una primitiva de muy bajo nivel para su uso directo

Linux Namespaces

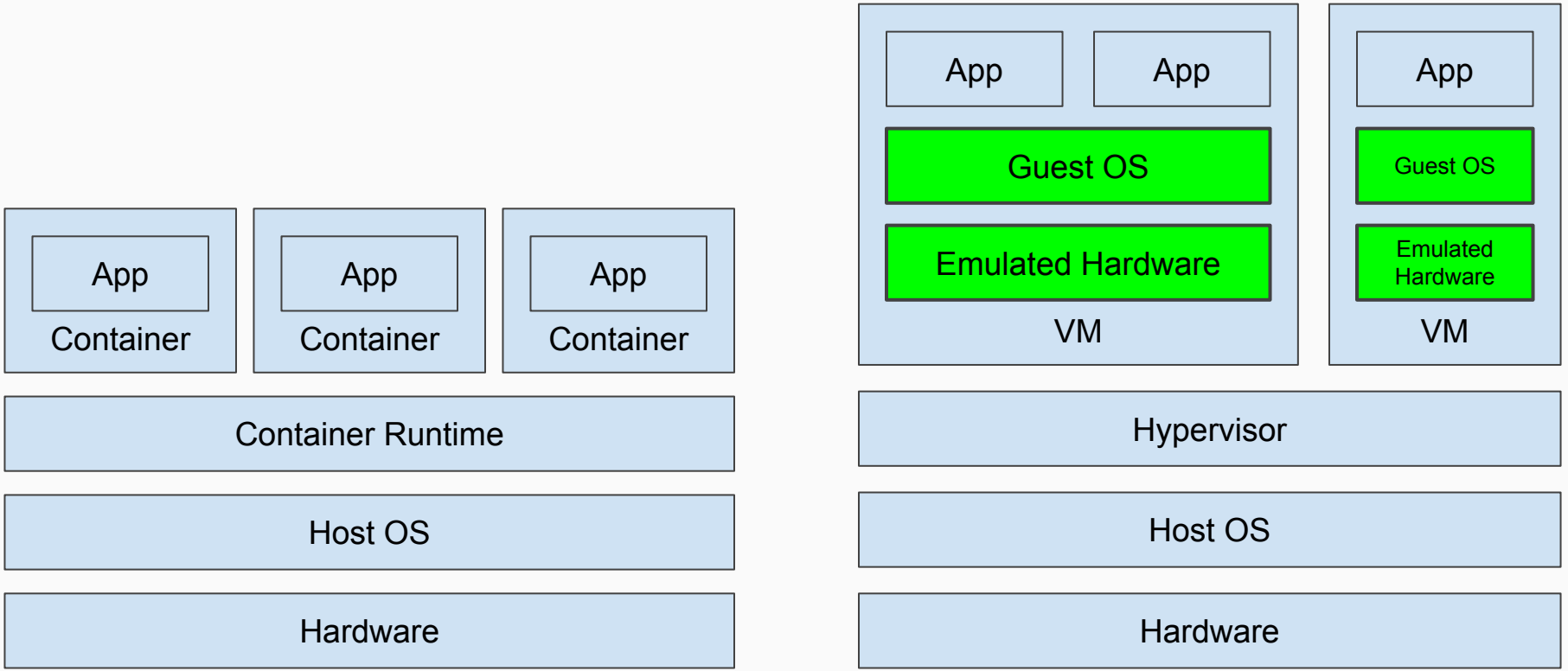
Mostrar todos los procesos:

```
$ ps -e

  PID TTY          TIME CMD
    1 ?        00:00:33 systemd
    2 ?        00:00:00 kthreadd
    4 ?        00:00:00 kworker/0:0H
    6 ?        00:00:00 mm_percpu_wq
    ...
```

Mostrar todos los procesos, aislando el namespace pid:

```
$ unshare --pid --mount-proc --fork  ps -e

  PID TTY          TIME CMD
    1 pts/0    00:00:00 ps
```

Containers vs Virtual Machines



¿Qué es Docker?

Docker es una herramienta que automatiza el despliegue de aplicaciones aisladas dentro de contenedores

Docker

Conceptos centrales que administra Docker:

➔ **container**: Grupo aislado de procesos

➔ **image**: Template usado para crear contenedores
(Sistema de archivos + metadata)

➔ **Dockerfile**: Archivo con instrucciones para construir una imagen

¿Qué brinda Docker?

- **Consistencia**
  Cualquier imagen se obtiene, inicia y configura de la misma manera

- **Replicabilidad**
  Los contenedores basados en una misma imagen son inicialmente iguales

- **Aislamiento**
  Se puede controlar la interacción entre la aplicación en un contenedor y el mundo exterior (en ambas direcciones)

Docker

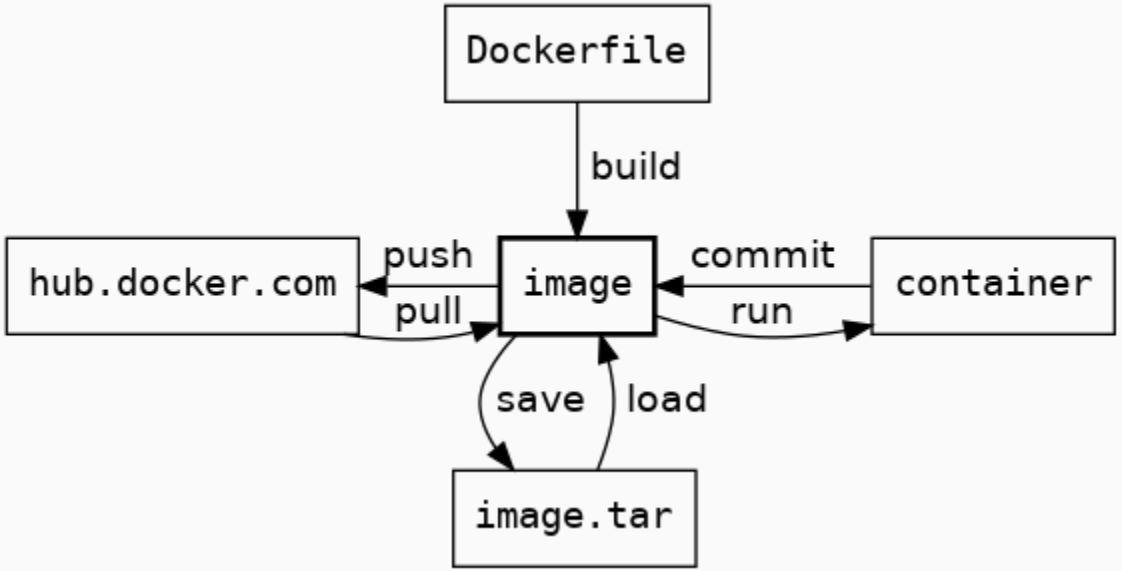
```
$ docker run hello-world

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.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

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/
```

Contenedores Livianos

El ecosistema Docker asume que los contenedores son "livianos":

- ➔ Dedicados a un único propósito
  - ◆ Ejecutan un comando
  - ◆ No incluyen dependencias innecesarias
- ➔ Mantienen el estado mínimo necesario
  - ◆ "Mínimo necesario" no significa "Poco". Por ejemplo, bases de datos
- ➔ Pueden encender/apagar rápidamente
  - ◆ Útil para escalar horizontalmente

Docker



No hay forma de compartir un contenedor en sí, pero se pueden compartir:

- Imágenes en repositorios como hub.docker.com
- Imágenes exportadas a archivos comprimido
- El Dockerfile que generó la imagen (Aunque eso no garantiza generar una imagen idéntica)

Docker

Otros elementos que maneja Docker:

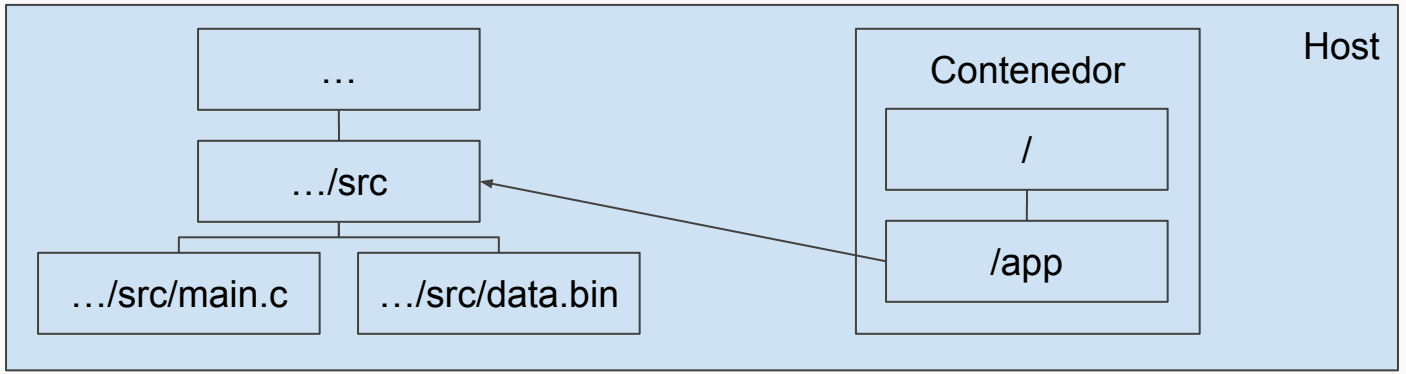
- ➔ **mounts**: Acceso a directorios del host
- ➔ **volumes**: Área donde se persisten datos

- ➔ **networks**: Interfaces de red internas
- ➔ **ports**: Forwardear puertos del host a puertos del contenedor

Docker

```
$ docker run -v host_dir_or_volume:container_dir ...
```

- Da acceso a un volumen/mount en un directorio del contenedor
- Opcionalmente, se puede marcar Read Only (`:ro`)

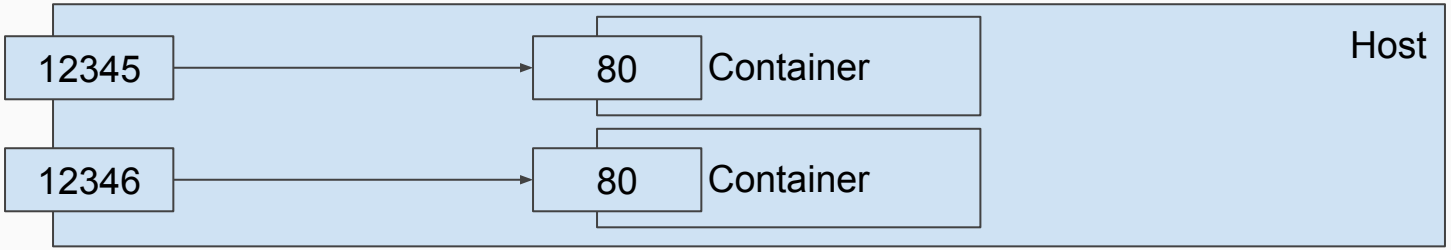


Docker

```
$ docker run -p external_port:internal_port ...
```

- Forwardea paquetes de red recibidos en `external_port` del host, a `internal_port` del contenedor
- Puede especificar ip en la que se reciben los paquetes

```
-p 127.0.0.1:12345:80
```



Docker

```
$ mkdir nginx_files
$ echo "Hello World" > nginx_files/index.html
```

¿Qué debe suceder aquí?

```
$ curl localhost:12345
Hello World
```

Docker

```
$ mkdir nginx_files
$ echo "Hello World" > nginx_files/index.html
```

```
$ docker run -v "$PWD/nginx_files:/usr/share/nginx/html:ro" \
             -p 12345:80 \
             nginx:1.23
```

```
$ curl localhost:12345
Hello World
```

Docker

Con la misma facilidad que pudimos iniciar un servidor web, podríamos haber iniciado cualquier aplicación dockerizada

Conociendo solo mapeos de puertos, almacenamiento y variables de entorno

➔ Vista fisica de 4+1

Docker

Algunos comandos comunes:

- docker logs
- Listados de objetos
  - docker image ls
  - docker container ls
  - docker volume ls
- Eliminar objetos
  - docker image rm
  - docker container rm
  - docker volume rm
- Control de contenedores
  - docker run <image>
  - docker stop <container>
  - docker start <container>
  - docker kill <container>

El flag más importante:

--help

Lista subcomandos y flags

Dockerfile

Docker

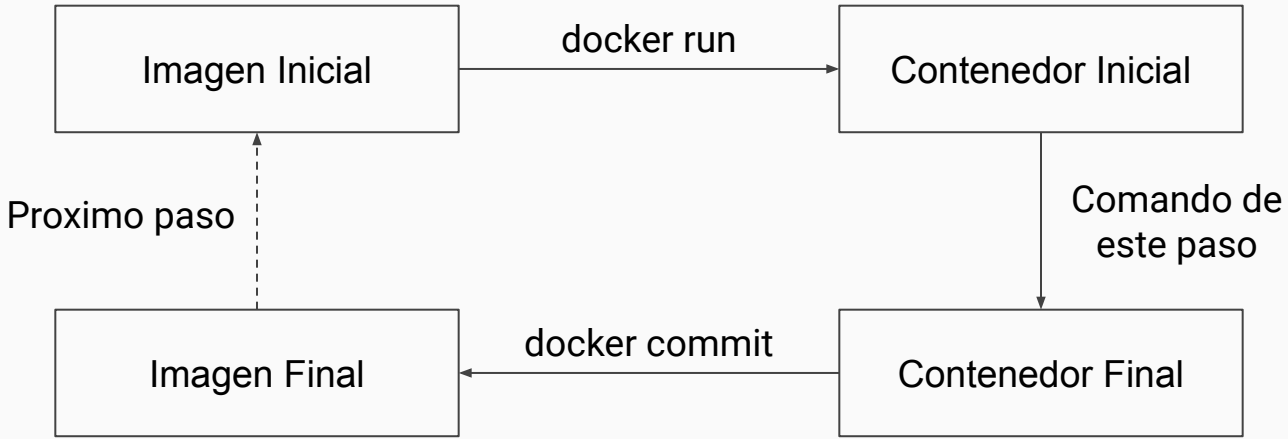
```
# Dockerfile

FROM node:16

WORKDIR /app
COPY . .

RUN npm ci

CMD ["node", "server.js"]
```

Dockerfile simplificado de una aplicación interpretada (Javascript en Node)

- Comienza con una imagen oficial node
- Selecciona un directorio del contenedor y copia todos los archivos del directorio actual del host
- Instala dependencias
- Finalmente, indica qué comando se ejecuta al iniciar un contenedor con esta imagen

Dockerfile

Serie de pasos para construir una imagen.



docker history <imagen>: Muestra cada paso que produjo una imagen

Dockerfile

Las imágenes se almacenan como una referencia a la imagen anterior y una lista de diferencias, llamada **layer**

- ➔ Cada layer es inmutable
- ➔ Las referencias son de un layer a su predecesor
  - ◆ Dos builds pueden compartir layers hasta la primer diferencia
  - ◆ Un archivo agregado en un layer y borrado en otro layer sigue siendo accesible
  - ◆ Imágenes ya construidas sirven como una cache

Dockerfile

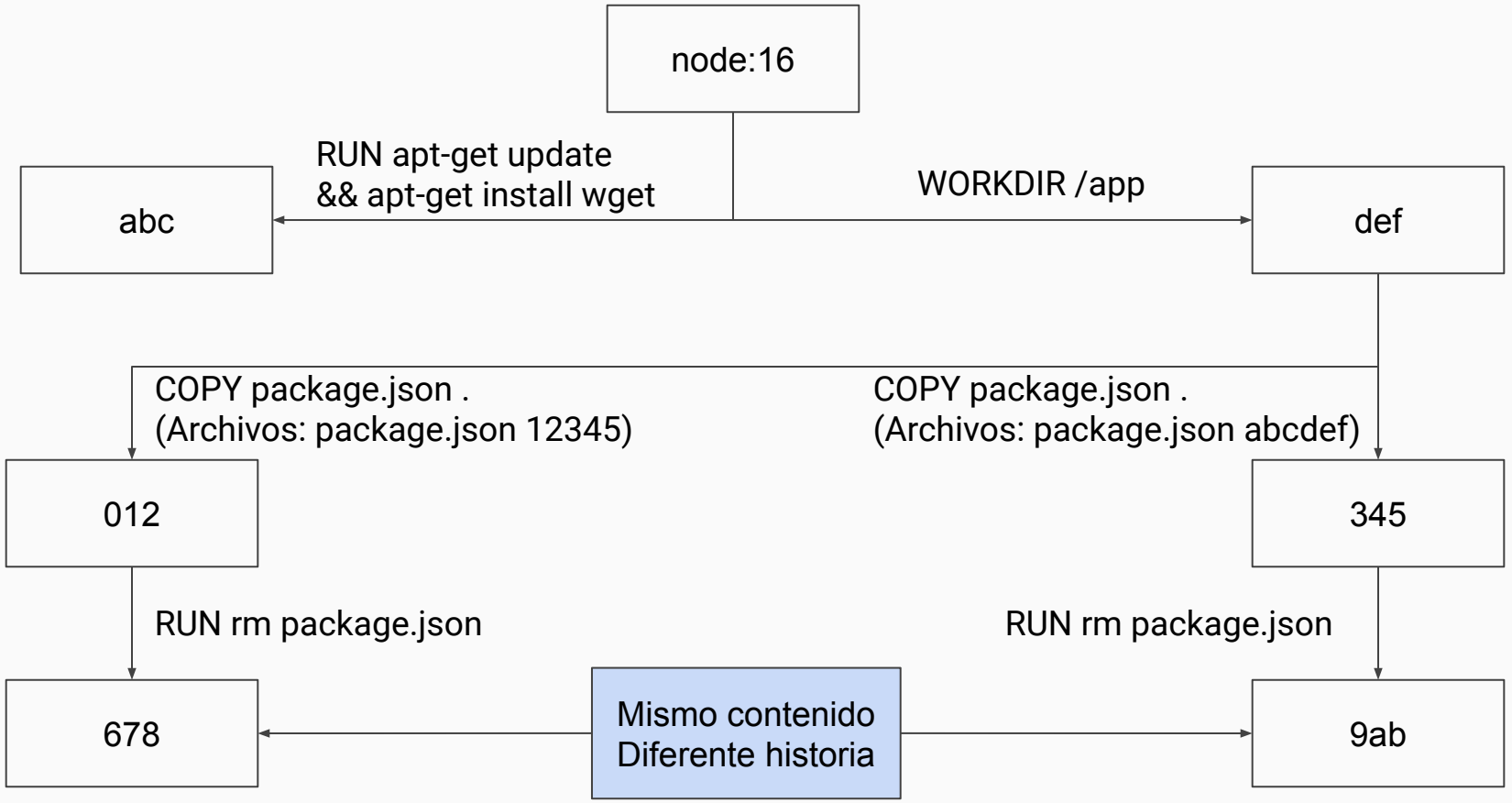
Para reuso, se asume que cada layer depende solo de:

- Imagen anterior
- El comando ejecutado
- Los archivos usados del directorio del Dockerfile (el contexto)
  - `.dockerignore`: Excluir archivos del contexto (git, node_modules, etc)

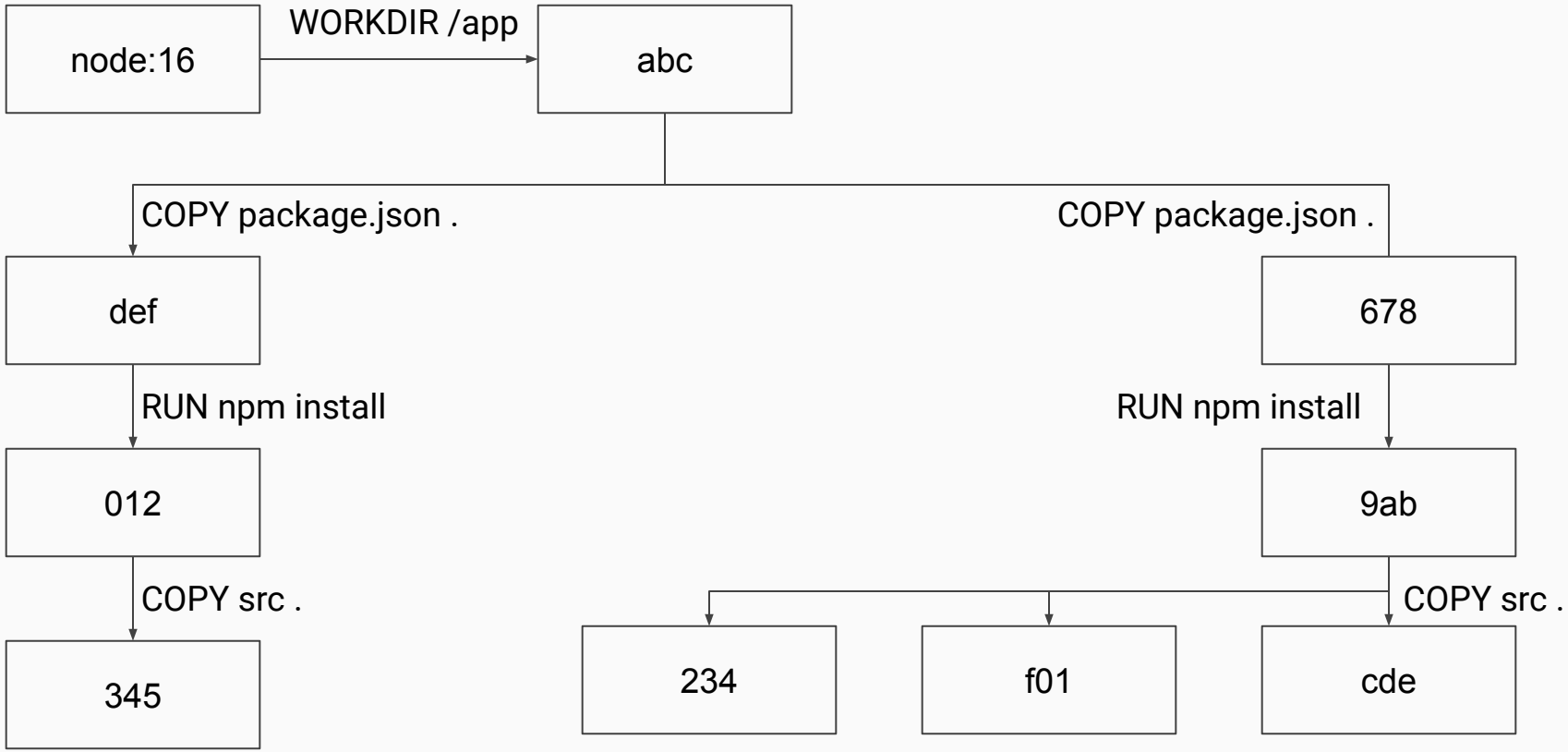
Se ignoran otras **entradas implícitas**:

- Comunicación con internet
- Aleatoriedad
- Pasaje del tiempo
- Etc

Dockerfile


node:16
RUN apt-get update
&& apt-get install wget
abc
WORKDIR /app
def
COPY package.json .
(Archivos: package.json 12345)
COPY package.json .
(Archivos: package.json abcdef)
012
345
RUN rm package.json
RUN rm package.json
678
Mismo contenido
Diferente historia
9ab

Dockerfile


node:16
WORKDIR /app
abc
COPY package.json .
COPY package.json .
def
678
RUN npm install
RUN npm install
012
9ab
COPY src .
COPY src .
345
234
f01
cde

Dockerfile

```
# Dockerfile

FROM node:12-slim
WORKDIR /app

COPY ./package.json .
COPY ./package-lock.json .
RUN npm ci

COPY . .
CMD ["node", "server.js"]
```

Dockerfile típico de una aplicación interpretada (Javascript en Node)

- Copia solo los archivos necesarios para instalar dependencias, que cambian infrecuentemente

- "RUN npm ci" suele ver:
  - Misma imagen generada por COPY
  - Mismo comando a ejecutar

  Intenta usar un layer cacheado.

- Finalmente, copia el código de aplicación

Ahorrar una ejecución de "RUN npm ci" puede no parecer mucho, pero debe multiplicarse por todos los builds que haga el equipo de desarrollo

Dockerfile

```
# Dockerfile
# Ejemplo de un error

RUN apt-get update
RUN apt-get install dep1 dep2 ... depN



# En el futuro

RUN apt-get update
RUN apt-get install dep1 dep2 ... depN depN+1



# Solucion
RUN apt-get update && \
    apt-get install dep1 dep2 ... depN depN+1
```

Error típico de cache

- "`RUN apt-get update`" obtiene versiones disponibles en servidores
- "`RUN apt-get install`" instala las versiones obtenidas
- Agregar una dependencia invalida la cache de "`RUN apt-get install`", intenta obtener versiones sin verificar si están disponibles
- Solución: Ejecutar ambos comandos en una sola invocación
- apt-get lanza error inmediatamente, otros comandos suelen **fallar silenciosamente**

Imagenes multi-etapa

Separa la construcción de una imagen en diferentes etapas

- Cada etapa
  - Parte de una nueva imagen base
  - Se puede construir en paralelo con otras etapas
- Permite copiar archivos de una etapa a otra
- Usado para no incluir compiladores/linters/etc en runtime

Dockerfile

```
# Dockerfile
# Stage 0: Create jar with JDK
FROM maven:3.8.3-openjdk-17-slim AS build

WORKDIR /app
COPY pom.xml .
RUN mvn dependency:resolve \
        dependency:resolve-plugins

COPY . .
RUN mvn package -DskipTests



# Stage 1: Run jar with JRE
FROM eclipse-temurin:17-jre-focal AS deploy

COPY --from=build /app/target/main.jar /app
CMD ["java", "-jar", "/app/main.jar"]
```

Dockerfile típico de una aplicación compilada (Java Maven)

- Etapas separadas de compilación y despliegue
- Despliega la aplicación en una imagen con menos capacidades que las necesarias para compilarla
  - Menos recursos consumidos en servidores
  - Reduce la superficie vulnerable de los contenedores

Dockerfile

```
# Dockerfile
# Stage 0: Create js bundle

FROM node:12-slim AS build
WORKDIR /app

COPY ./package.json ./package-lock.json ./
RUN npm ci

COPY . .
RUN npm run build



# Stage 1: Web server

FROM nginx:1.23 AS deploy
COPY --from=build /app/build/
                  /usr/share/nginx/html/
```

Dockerfile típico de una aplicación compilada (React en Node)

- Etapas separadas de compilación y despliegue
- Despliega la aplicación en una imagen con menos capacidades que las necesarias para compilarla
  - Menos recursos consumidos en servidores
  - Reduce la superficie vulnerable de los contenedores

Seguridad

Seguridad

Un detalle con sistemas de archivos unix:

- El control de acceso a archivos es por id de usuario
  - Excepto root (uid = 0), que tiene acceso a todo
- El sistema de archivos solo almacena números de id de usuario
  - No distingue qué sistema originó el id de usuario
- Un contenedor puede ejecutar con cualquier id de usuario que quiera

➔ El único control de acceso a un volumen es ser solo lectura

Seguridad

Cualquier usuario con permiso de crear contenedores puede escalar a ser root

```
docker run -it -v /:/host ubuntu
```

Orquestadores

Orquestadores

Un **orquestador** coordina el uso de múltiples contenedores para implementar una única aplicación, brindando:

- Configuración de contenedores heterogéneos
- Distribución
- Balanceo de carga
- Tolerancia a fallos

Algunos comunes: docker-compose, kubernetes

Docker-compose

Anteriormente usamos directamente:

```
$ docker run -v "./nginx_html:/usr/share/nginx/html:ro" -p 12345:80 nginx:1.23
```

Su docker-compose.yml equivalente:

```
version: "3"

services:
    webserver:
        image: nginx:1.23
        volumes:
            - "./nginx_html:/usr/share/nginx/html:ro"
        ports:
            - "12345:80"
```

Para iniciar: `docker-compose up -d`
Para detener: `docker-compose down`

Docker-compose

```
# docker-compose.yml

version: "3"

volumes:
    db_persist:

services:
    db:
        image: mysql:8.0
        ports:
            - "12345:3306"
        volumes:
            - db_persist:/var/lib/mysql
        environment:
            MYSQL_USER: "${DB_USER}"
            MYSQL_PASSWORD: "${DB_PASSWORD}"
```

Usualmente, el cliente es parte del sistema y la base de datos está oculta, ver próximo slide

docker-compose.yml de una base de datos:

- Indica a docker-compose que gestione un volumen llamado "db_persist"
- La imagen mysql recibe dos variables de entorno, que provienen de docker-compose
- docker-compose completa las variables con valores que recibe:
    - Como variables de entorno propias
    - En un archivo indicado por --env-file (o .env si no se indica)

Orquestadores

Usamos el archivo .env para separar configuración:

- Del contexto donde se despliega
  - Por ejemplo: Puertos, URLs, API keys
  - Variables en .env

- Interna al sistema
  - Cosas configurables para los contenedores que lo componen, pero que no son visibles desde fuera del sistema
  - Por ejemplo: Puertos/hosts usados para comunicación interna
  - Hardcodeado en docker-compose.yml, posiblemente deduplicado con YAML Anchors

Docker-compose

```
# docker-compose.yml

version: "3"

volumes:
    db_persist:
services:
    db:
        image: mysql:8.0
        volumes:
            - db_persist:/var/lib/mysql
        environment:
            MYSQL_USER: "${DB_USER}"
            MYSQL_PASSWORD: "${DB_PASSWORD}"
    api:
        depends_on:
            db:
                condition: service_healthy
        image: my_awesome_api:0.1
        ports:
            - "12345:80"
        environment:
            DATABASE_HOST: db
            MYSQL_USER: "${DB_USER}"
            MYSQL_PASSWORD: "${DB_PASSWORD}"
```

docker-compose.yml típico de una api:

- El servicio de la base de datos no expone puertos al exterior
- El servicio de la api recibe una variable de entorno indicando cómo conectarse a la base de datos.

  Docker-compose configura el archivo hosts, tal que conectarse a "db" es equivalente a conectarse a la ip correspondiente
- Hay un depends_on, de api a db. Esto indica a docker-compose el orden para activar los contenedores

Docker-compose

```
# Partes de docker-compose.yml

db:
   image: mysql:8.0

api:
   build: ./src/api
   image: my_awesome_api:0.1

frontend:
   build: ./src/frontend

ingress:
   build:
      dockerfile: ./ingress.Dockerfile
      context: .
```

Para monorepos, donde se almacenan múltiples servicios dentro de una jerarquía de archivos, existen otras formas de definir cómo obtener una imagen:

- Por nombre de imagen
- Indicando un directorio (docker-compose intenta hacer docker build de Dockerfile)
  - Con nombre autogenerado, a menos que se especifique uno usando "image"
  - Solo se intenta hacer el build:
    - Si la imagen no existe
    - Con docker-compose build
    - Con docker-compose up --build
- Indicando ubicación de un Dockerfile y del contexto de archivos a usar

Docker-compose

Mala practica extremadamente comun: container_name

```
services:
    api:
        container_name: api
```

- Fuerza que todas las instancias del servicio se llamen api
  - Colisión de nombres con otros proyectos
  - Colisión de nombres entre múltiples instancias
- Por defecto, docker-compose asigna nombres sin colisiones: proyecto_api_1, proyecto_api_2, ..., proyecto_api_N

CI/CD

CI/CD

Dos formas adicionales de determinar el entorno en que se ejecuta un build:

- ➔ Build crea una imagen docker
  docker push a un registro

- ➔ Ejecutar build dentro de un contenedor
  Extraer artefactos del contenedor

En ambos casos, docker hace a los builds independientes del runner

CI/CD

Mejores despliegues:

- ➔ Producir imágenes en build
- ➔ Subir imágenes a registro privado (GitLab, GitHub, etc)
- ➔ Usar mismas imágenes en diversos entornos
  - ◆ Variables de entorno para controlar variaciones

- ➔ Ops puede manejar componentes de manera uniforme
- ➔ Mínimas dependencias en servidor: Docker (+ orquestador)

¿Preguntas?

Recursos

- Documentación de Docker
- Docker Hub

- Referencia del formato Dockerfile
- Referencia del formato Compose

¡Gracias!