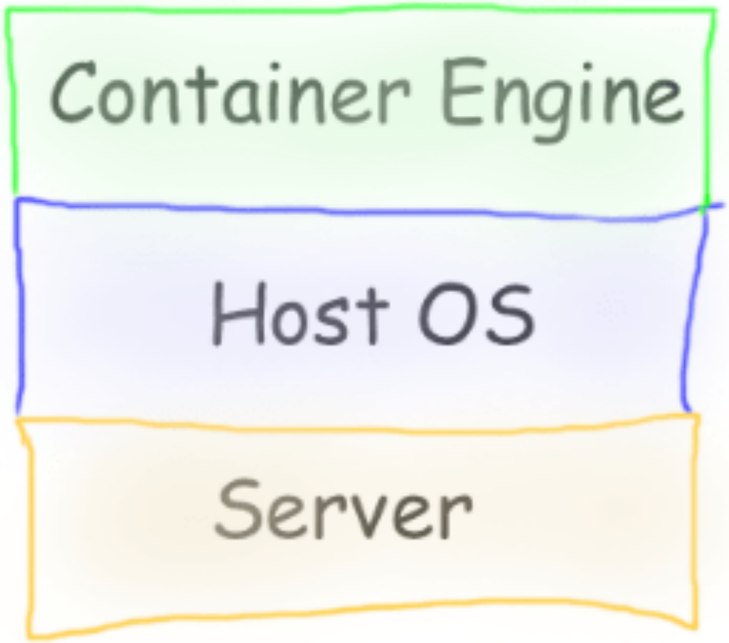
Desarrollo de Microservices Cloud Native

Contenedores

Contenedores

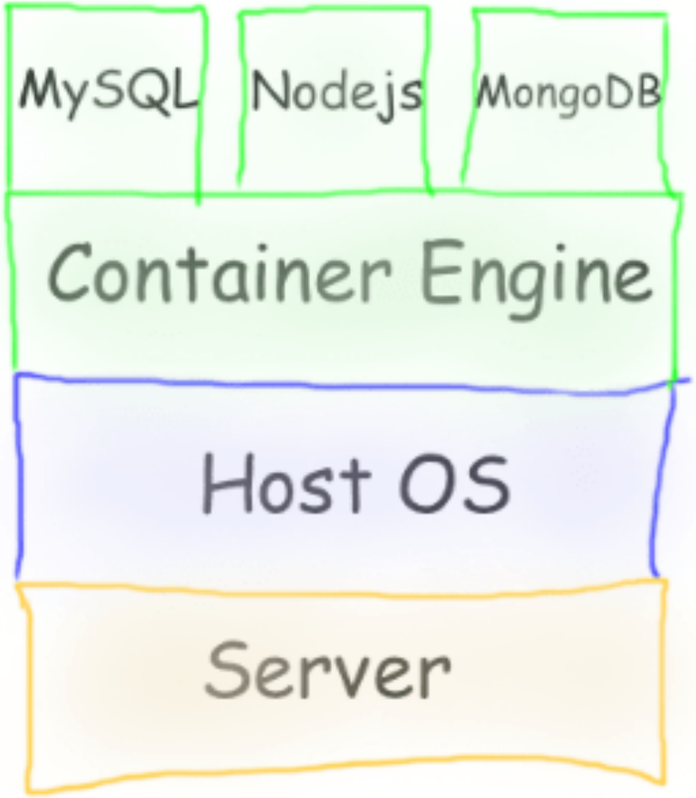
- Los contenedores son procesos aislados que comparten recursos con su host y a diferencia de las VM, no virtualizan el hardware y no necesitan un sistema operativo huesped.
- Una de las mayores diferencias entre los Contenedores y las VM es que los Contenedores comparten recursos con otros Contenedores en el mismo host. Esto nos brinda automáticamente más rendimiento que las máquinas virtuales, ya que no tenemos un sistema operativo huésped para cada contenedor.

En lugar de tener un hipervisor, ahora tenemos un Container Engine



TECH IS NOW

Aplicaciones corriendo en contenedores



TECH IS NOW

Beneficios

- Proceso aislado
  - Los contenedores son entornos que se ejecutarán mediante procesos aislados pero que pueden compartir recursos con otros contenedores en el mismo host
- Archivos montados
  - Los contenedores nos permiten montar archivos y recursos desde el interior del contenedor hacia el exterior
- Proceso liviano
  - Los contenedores no se ejecutan en un sistema operativo huesped, por lo que su proceso es liviano con un mejor rendimiento y puede iniciar el contenedor en cuestión de segundos.

Inconvenientes de los contenedores

- Mismo sistema operativo host
  - Puede encontrarse en una situación en la que cada aplicación requiere un sistema operativo específico y es más fácil de lograr con las máquinas virtuales, ya que podemos tener un sistema operativo huésped diferente.
- Temas de seguridad
  - Los contenedores son procesos aislados que tienen acceso directo a algunos espacios de nombres importantes, como el nombre de host, las redes y la memoria compartida. ¡Tu contenedor puede usarse para hacer cosas malas más fácilmente! Por supuesto, puede controlar a su usuario root, crear algunas barreras, pero debe preocuparse por ello.

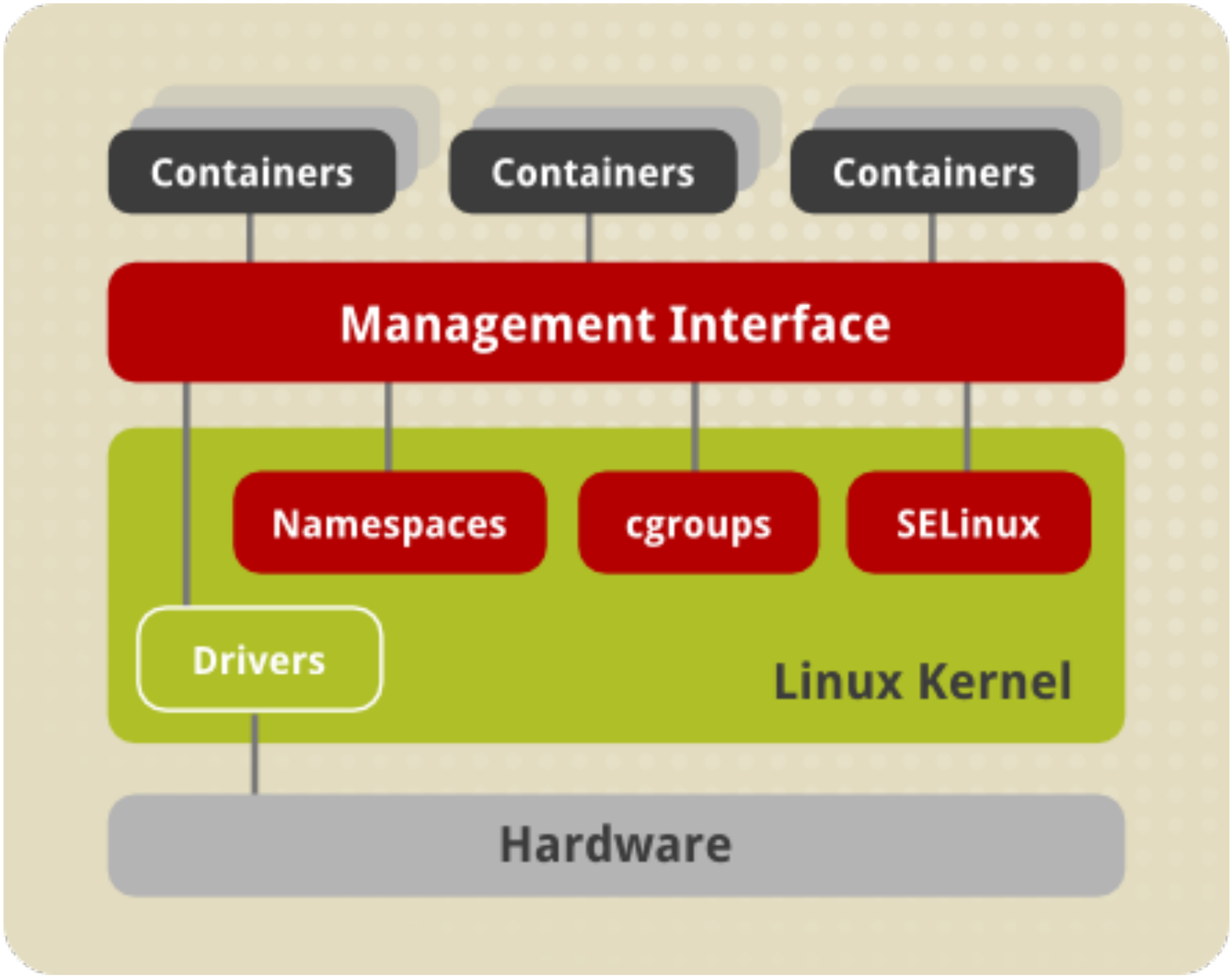
LXC (Linux Containers)

LXC
Introducción

- Es una forma de virtualización muy ligera
- Solo funcionan en Linux
- De forma similar que Jails de FreeBSD y Solaris Zones, LXC son ambientes de ejecución autocontenidos.
- Tiene sus propios recursos aislados de CPU, memoria, I/O y recursos de red (dirección IP).
- Deben correr sobre un ambiente huésped (host)
- Un contenedor comparte recursos de hardware con otros contenedores sin necesidad de apartarlos
- Comparte el kernel del SO huésped.

LXC
Introducción

- El resultado de usar LXC es que se siente como una máquina virtual, pero que elimina toda la carga y costo de arranque del sistema operativo huésped.
- En sistemas grandes, el uso de VMs implica duplicar instancias del mismo SO y muchos volúmenes de arranque redundantes.
- Debido a que LXC son más eficientes y ligeros comparados a VMs, se pueden ejecutar más contenedores que VMs en el mismo hardware.
- En un ambiente que implique Web-scale, LXC son más atractivos comparados con la virtualización de servidores tradicionales.


Containers
Containers
Containers
Management Interface
Namespaces
cgroups
SELinux
Drivers
Linux Kernel
Hardware

LXC Fundamentos

Control Groups
Linux cgroups

- Desarrollados originalmente por Google, se encargan del aislamiento y uso de los recursos del sistema, tales como CPU y memoria, para un grupo de procesos.

- Se pueden poner procesos dentro de un group para limitar el consumo de CPU y memoria.

Linux namespaces

- Namespaces se encargan del aislamiento de recursos para un único proceso, a su vez que cgroups lo hace para un grupo de procesos.

- Un namespace envuelve un recurso del sistema global en una abstracción que hace parecer al proceso dentro de un namespace que permite tener su propia instancia aislada. Los cambios que ocurren en el recurso global son visibles a otros procesos que son miembros del namespace, pero son invisibles para otros procesos.

LXC - Beneficios

- Los contenedores desacoplan las aplicaciones del sistema operativo; esto implica que los usuarios pueden tener un Linux muy limpio y correr cualquier otra aplicación en contenedores aislados.

- Además de que el sistema operativo está también abstraído de los contenedores, se pueden mover los contenedores en cualquier Linux que soporte el entorno ejecución del contenedor.

Docker

Docker - Intro

- Inició como un proyecto para construir contenedores LXC de aplicaciones individuales (single-applications).
- Aportó mejoras significantes a LXC para hacer contenedores más portables y flexibles de usar.
- Los contenedores de Docker te permiten desplegar, replicar, mover y respaldar una carga de trabajo de forma más fácil que usando MVs.
- Docker popularizó LXC en el día a día debido a su simpleza.

Diferencias de Docker y LXC

- Al inicio Docker intentaba ser una abstracción de LXC, pero evolucionó a tener su propio entorno de ejecución de contenedores.
  - Usa LXC como biblioteca internamente.

Docker
Single process

- Si necesitamos N procesos, debemos usar N contenedores.
- Ejemplo, típica webapp, requiere AppServer, DBServer y WebServer
  - 3 contenedores
- El beneficio principal es que se gana granularidad.
- Las actualizaciones se aplican por contenedor.

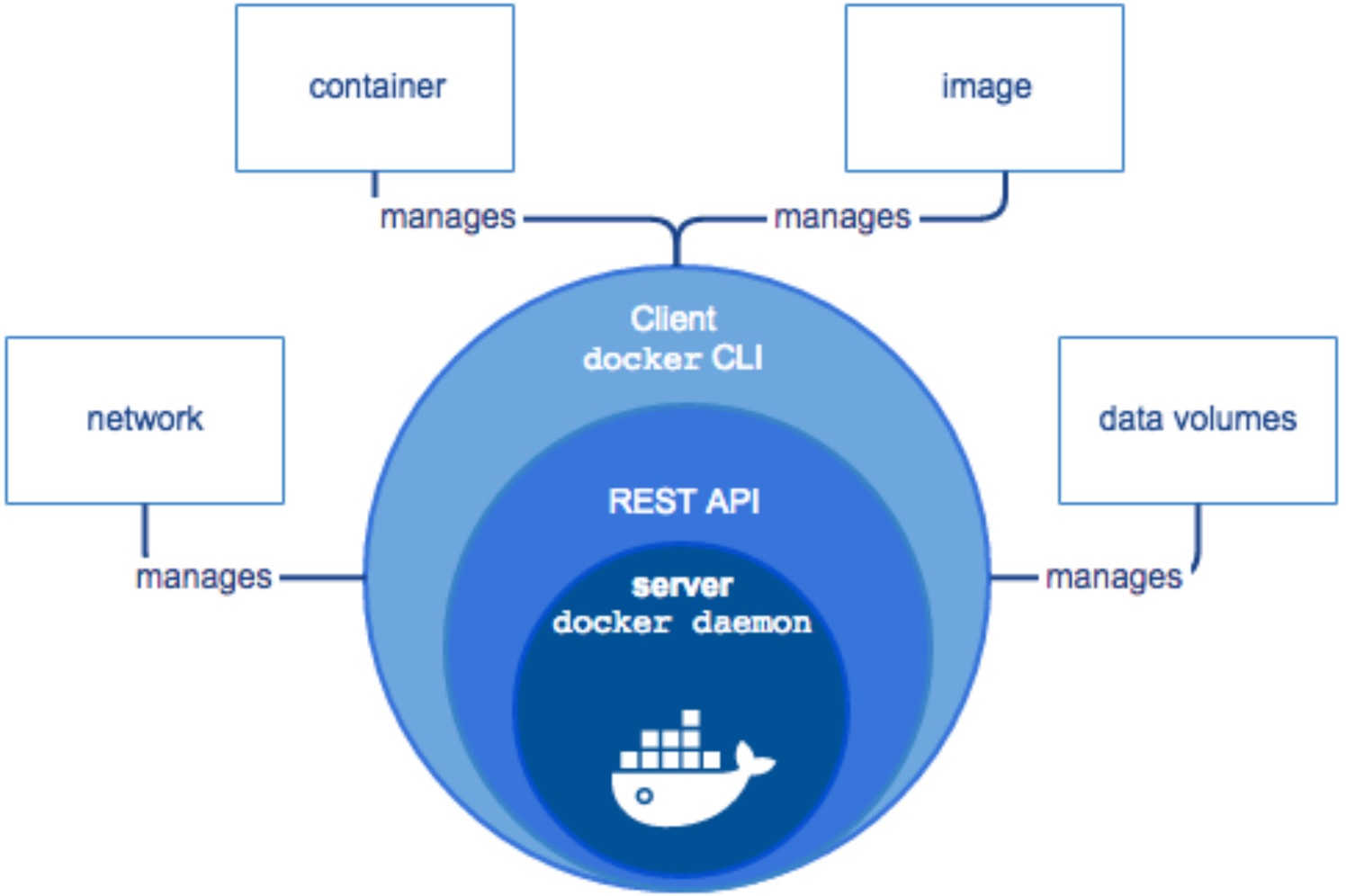
LXC
Multiple processes

- Con LXC se pueden ejecutar varios procesos.
- Debido a que soporta init-process
  - En un solo contenedor podríamos correr tantos procesos de servidores como deseemos.

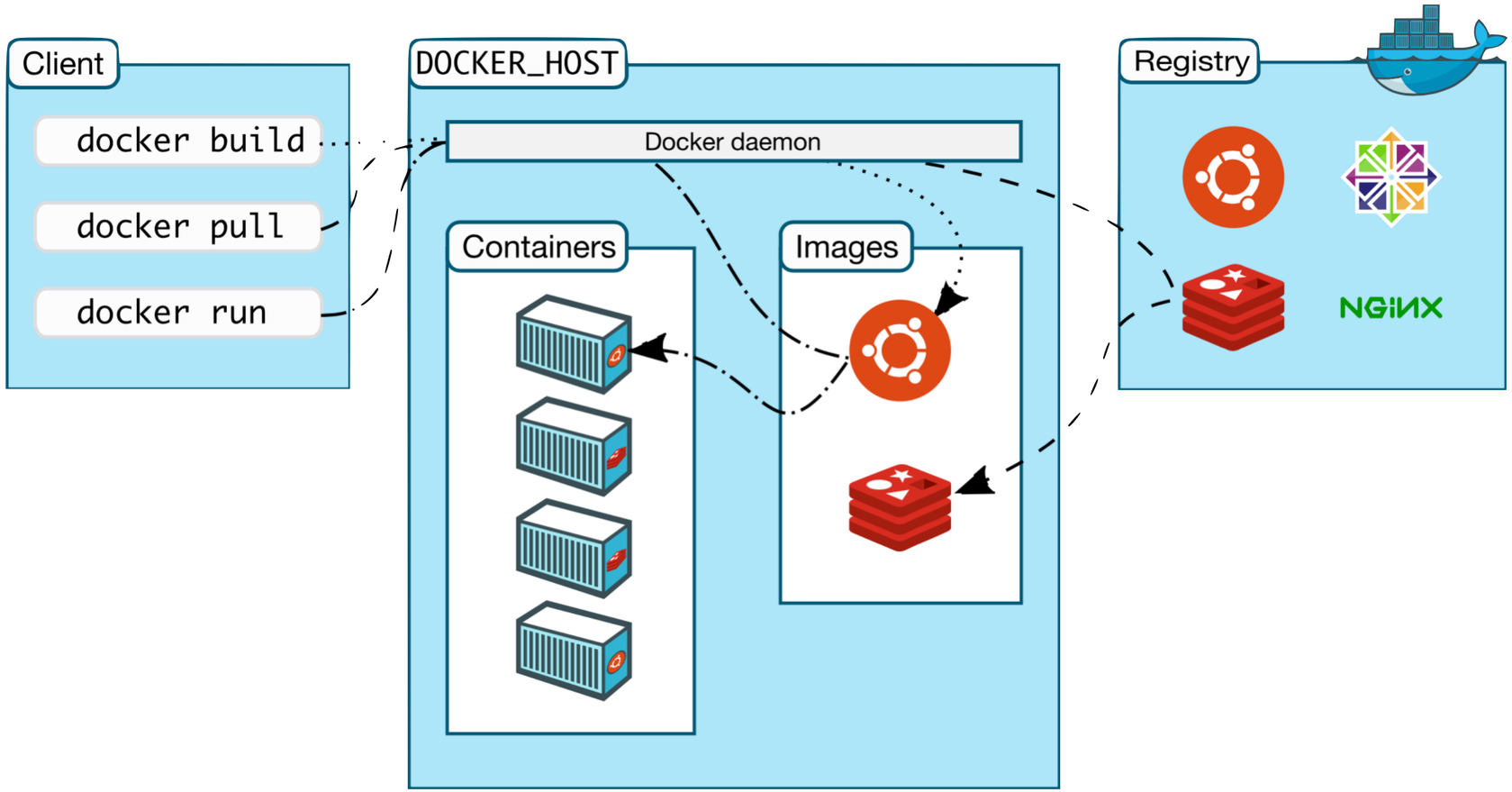
Docker
Stateless

- Docker no soporta almacenamiento persistente.
- Para almacenamiento se usa el huésped a través de volúmenes de Docker.
- Docker proporciona “layers” de solo lectura.
- Cada cambio genera un nuevo layer que puede ser guardado, lo que genera una nueva versión de la imagen del contenedor.
- Se puede hacer rollback hasta cierto “commit” de los layers.

Docker Engine



Arquitectura de Docker



Componentes

- **Docker daemon**
  - Corre en la máquina huésped. No se interactúa directamente, sino a través del Docker client.
- **Docker client**
  - Interface primaria de Docker, acepta comandos de usuario y se comunica de ida y vuelta con el Docker daemon.
- **Docker images**
  - Plantilla de solo lectura, se usan para crear contenedores de Docker. Una imagen puede tener una versión especifica de Linux con tu aplicación instalada. Se pueden usar imágenes ya creadas o crear una personalizada.
- **Docker registries**
  - Lugar donde se almacenan las imágenes. Pueden ser públicos o privados.
  - DockerHub (https://hub.docker.com)
- **Docker containers**
  - Son similares a un directorio. Almacena todo lo que una aplicación requiere para ejecutarse. Se crean a partir de una imagen. Pueden ser arrancados, iniciados, detenidos, movidos y borrados. Cada contenedor es una plataforma de aplicaciones aislada y segura.

Docker images

- Cada imagen consiste en un conjunto de ***layers***.
- Docker usa ***union file systems*** para combinar las capas en una sola imagen.
- ***Union file system*** permiten que archivos y directorios de sistemas de archivos separados (***branches***), transparentemente sobrepuestos, formen un único y coherente ***file system***.
- Cuando se cambia algo en la imagen, se construye un nuevo ***layer***, por lo tanto solo se debe actualizar los nuevos ***layers*** en lugar de toda la imagen.

Docker images

- Cada imagen inicia de una imagen base, por ejemplo ***ubuntu*** o ***fedora***.
- Las imágenes se crean a partir de estas imágenes base, usando pasos de un conjunto simple y sencillo de ***instructions***.
- Cada ***instruction*** crea un nuevo ***layer*** en la imagen. Las ***instructions*** son acciones, ejemplo:
  - Ejecutar un comando
  - Agregar un archivo o directorio
  - Crear una variable de ambiente
  - Qué proceso se va a ejecutar cuando se lance un contenedor de la imagen.
- Las ***instructions*** se almacenan en un archivo llamado ***Dockerfile***. Un ***Dockerfile*** es un script en archivo de texto que contiene las ***instructions*** y ***commands*** para construir la imagen a partir de la imagen base.
- Cuando se solicita construir la imagen, Docker lee el ***Dockerfile*** y ejecuta las ***instructions*** y regresa la imagen final.

Docker containers

- Consiste en un sistema operativo, archivos agregados por el usuario y meta-data. Se construyen a partir de una imagen.

- La imagen indica lo que el contenedor almacena, el proceso que va a ejecutar. Es de solo lectura.

Instalando Docker

- Windows
  - https://docs.docker.com/docker-for-windows/
- MacOS
  - https://docs.docker.com/docker-for-mac/
- Linux
  - https://docs.docker.com/engine/installation/linux/

Lanzar un contenedor

```
$ docker run -i -t ubuntu /bin/bash
```

- Obtiene la imagen ubuntu. Se puede obtener localmente o remotamente del DockerHub.
- Crea el contenedor con la imagen obtenida.
- Asigna el ***filesystem*** y monta el ***layer*** de solo lectura.
- Asigna la interface de ***network / bridge***: Esto permite que el contenedor pueda hablar con el localhost
- Configura la dirección IP: Dirección IP disponible de un ***pool***.
- Ejecuta el proceso que se especificó.
- Captura la salida de la aplicación. Conecta y registra la entrada, salida y error estándar para que se observe cómo se ejecuta el proceso.

Imágenes y contenedores

```
$ 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.
 3. The Docker daemon created a new container from that image which runs the exec
 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 Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/

$
```

Docker - comandos

Comando	Intención
run [OPTS] IMAGE [COMMAND] [ARG...]	Ejecuta un contenedor
ps	Lista los contenedores
logs [OPTIONS] CONTAINER	Obtiene los logs
stop [OPTS] CONTAINER [CONTAINER...]	Detiene el contenedor
rm [OPTS] CONTAINER [CONTAINER...]	Remueve uno o mas contenedores
images	Lista las imágenes
pull [OPTIONS] NAME[:TAG|@DIGEST]	Obtiene imágenes remotas
rmi [OPTIONS] IMAGE [IMAGE...]	Remueve imágenes

Práctica Spring Boot y JPA

TECH IS NOW

Dockerizando Spring Boot clase03/01_intro_docker

Pasos

- ./gradlew clean build
  - –gradlew.bat clean build
- cd build/libs
- docker build -t introdocker:v1 .
  - –docker build -t introdocker:v1 .
- docker images
- **Ejecutar el contenedor**
  - **DETACH**: docker run -d -p 8080:8080 --name demo1 introdocker:v1
    - -d: detach mode
    - -p: puertoLocal:puertoContenedor
    - --name: nombre del contenedor (único) completamente opcional
  - docker run -p 8080:8080 --name demo1 introdocker:v1
- docker ps
- docker stop CONTAINER

Docker Compose

Docker Compose

- Con Compose podemos definir aplicaciones que usen múltiples contenedores de Docker.
- En un archivo de configuración se definen los servicios que usa la aplicación.
- Con una sencilla instrucción en el CLI, se pueden arrancar todos a la vez.

Docker Compose
Uso en tres pasos

1. Se crea un ***Dockerfile*** con la configuración de la aplicación.
2. Se definen los servicios que se utilizan en un archivo llamado ***docker-compose.yml***.
3. Se ejecuta ***docker-compose up*** para iniciar toda la aplicación.

docker-compose.yml

```
version: '2'
services:
  app:
    build: app
    environment:
        - SPRING_RABBITMQ_HOST=rabbitmq
        - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/cacti_data
    ports:
     - "8080:8080"
    depends_on:
     - postgres
     - rabbitmq
  rabbitmq:
    image: rabbitmq
  postgres:
    build: postgres
    environment:
      - POSTGRES_PASSWORD=secret
    volumes:
      - ../../postgres:/var/lib/postgresql/data
```

Docker Compose features

Múltiples ambientes aislados en el mismo host

- Se usa un nombre de proyecto para aislar los ambientes.
  - Se pueden crear múltiples copias del mismo ambiente.
  - En un servidor de CI se puede usar para que no interfiera con otros trabajos
  - En un host compartido se pueden usar los mismos servicios en diferentes proyectos
- El nombre del proyecto por omisión es el nombre del directorio. Se puede modificar con el flag -p o con la variable de ambiente COMPOSE_PROJECT_NAME

Docker Compose features

- Almacena los volúmenes de datos que crean los contenedores
  - Se guardan todos los volúmenes que usan los servicios.
- Solo recrea contenedores que han cambiado.
  - La configuración es cacheada, cuando se reinicia un servicio que no ha cambiado, se rehúsan los contenedores.
- Se pueden usar variables en la configuración.
  - Esto permite que se personalicen a distintos ambientes o usuarios.

Docker Compose
clase03/02_docker-compose

Práctica

1. Crear una aplicación de Spring Boot con los siguientes starters:
   - web
   - data-jpa
   - data-rest
2. Añadir las bibliotecas
   - Postgres
   - flyway

Práctica

3.Agregar una clase de dominio sencilla

4.Agregar el repositorio de la clase de dominio

5.Renombrar el archivo de configuración Properties a formato YML

6.Configurar el DataSource para Postgres

```
spring.datasource:
  driverClassName: org.postgresql.Driver
  url: jdbc:postgresql://localhost:5432/c7_data
  username: c7
  password: c7
```

Práctica (Posible modelo)

```java
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;

@Entity
@Table(name = "people")
public class Person {
  @Id
  @GeneratedValue(generator = "uuid2")
  @GenericGenerator(name = "uuid2", strategy = "uuid2")
  @Column(updatable = false)
  private String id;

  @Version
  private Long version;

  private String name;

  private String email;
```

Práctica

7. Si se tiene instalado localmente Postgres crear el usuario y la base de datos. El password es **c7** (Opcional)

8. Correr el app. Seguro fallará (no está creada estructura de tablas)

```
$ createuser -d c7 -P
$ createdb -E utf8 -O c7 -W -U c7 c7_data
```

TECH IS NOW

Práctica

9.Agregar migraciones de Flyway. Adecuarla al modelo que se creó.

```
CREATE TABLE people (
  id       VARCHAR(255) PRIMARY KEY,
  version  BIGINT       NOT NULL,
  name     VARCHAR(255) NOT NULL,
  email    VARCHAR(255) NOT NULL
);
```

Práctica

- Las migraciones con Flyway se hacen con SQL nativo.
- Si el jar de Flyway está en el classpath, entonces Spring Boot automáticamente va a cargar las migraciones.
- Los archivos de las migraciones deben ubicarse en la carpeta:
  - src/main/resources/db/migration
- Los archivos deben llamarse V{N}__{nombre}.sql
  - N: número consecutivo, sirve para dar orden.
  - ATENCION: son 2 guiones bajos después de N

Dockerizar

src/main/docker/app/Dockerfile

```
FROM java:latest
VOLUME /tmp
ADD app.jar app.jar
RUN sh -c 'touch /app.jar'
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

¿Y la base de datos?

- Por default se configuró a localhost para conectarse a la base de datos.
- No hay Postgres en ese contenedor de Docker
- Aquí entra la ayuda de Docker Compose

Escribir el docker-compose.yml

- src/main/docker/docker-compose.yml

```yaml
version: '2'
services:
  app:
    build: app
    environment:
        # se sobrescribe la configuración para que apunte al contenedor, los
        # hostnames se agregan automáticamente en /etc/hosts de cada contenedor
       - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/c7
    ports:
     - "8080:8080"
    depends_on:
     - postgres
  postgres:
    image: postgres
```

Integrar Docker a Gradle
build.gradle

```
task prepareDocker(type: Copy) {
  from 'src/main/docker/'
  into 'build/libs'
}

task prepareApp(type: Copy) {
  from 'build/libs/app.jar'
  into 'build/libs/app'
}

prepareApp.dependsOn prepareDocker
build.dependsOn prepareApp
```

Script para facilitar el flujo de trabajo (compose.sh)

```
#!/usr/bin/env bash

# se construye la aplicación
./gradlew clean build

cd build/libs

# se levantan los contenedores configurados
docker-compose up --build
docker-compose stop

cd ../..
```

Intentar levantar

- No olvidar dar permisos de ejecución al script
- Fallará debido a que no se ha creado:
  - El usuario de la base de datos.
  - La base de datos en el contenedor de Postgres.

Configurar el contenedor de Postgres

https://hub.docker.com/_/postgres/

TECH IS NOW

- Crear la carpeta
  - src/main/docker/postgres
- Crear un Dockerfile
  - src/main/docker/postgres/Dockerfile

```
# Dockerfile
FROM postgres

# La imagen de Postgres permite agregar
# scripts personalizados
# Se pueden usar para crear estructuras iniciales o
# cargar datos.
COPY init-user-db.sh /docker-entrypoint-initdb.d/
```

init-user-db.sh

- Crear un archivo en:
  - src/main/docker/postgres/init-user-db.sh

```
#!/bin/bash
set -e

psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" <<-EOSQL

    CREATE USER c7 PASSWORD 'c7';
    CREATE DATABASE c7_data OWNER c7;
    GRANT ALL PRIVILEGES ON DATABASE c7_data TO c7;

EOSQL
```

Modificar el compose

```
version: '2'
services:
  app:
    build: app
    environment:
        # se sobreescribe la configuración para que apunte al contenedor los
        # hostnames se agregan automáticamente en /etc/hosts de cada contenedor
        - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/c7_data
    ports:
     - "8080:8080"
    depends_on:
     - postgres
 postgres:
    build: postgres
    environment:
        # le cambiamos el password al usuario de postgres
      - POSTGRES_PASSWORD=secret
```

Configurar Volúmenes

- La imagen de Postgres ya expone volúmenes.

```
version: '2'
services:
  app:
    build: app
    environment:
        # se sobreescribe la configuración para que apunte al contenedor los
        # hostnames se agregan automáticamente en /etc/hosts de cada contenedor
        - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/c7_data
    ports:
     - "8080:8080"
    depends_on:
     - postgres
 postgres:
    build: postgres
    environment:
      - POSTGRES_PASSWORD=secret
    volumes:
      - ../../postgres:/var/lib/postgresql/data
```

Race Condition

- En ocasiones los contenedores que dependen de otro contenedor, arrancan mucho mas rápido.
- No hay forma para arreglarlo oficialmente.
- Existen varias soluciones como:
  - https://github.com/vishnubob/wait-for-it

Dockerfile

```
FROM python:2-onbuild
RUN ["pip", "install", "pika"]
ADD start.sh /start.sh
CMD ["/start.sh"]
```

start.sh

```
#!/bin/bash
while ! nc -z rabbitmq 5672; do sleep 3; done
python rabbit.py
```

TECH IS NOW

Usando wait-for-it

- Descargar
  - https://raw.githubusercontent.com/vishnubob/wait-for-it/master/wait-for-it.sh
- Ponerlo en:
  - src/main/docker/app/wait-for-it.sh

Usando wait-for-it

- Crear archivo: src/main/docker/app/start.sh

```
#!/usr/bin/env bash

java -Djava.security.egd=file:/dev/./urandom -jar /app.jar
```

- Modificar src/main/docker/app/Dockerfile

```
FROM java:latest
VOLUME /tmp
ADD app.jar app.jar
ADD wait.sh /wait.sh
ADD start.sh /start.sh
RUN sh -c 'touch /app.jar'
ENTRYPOINT ["/wait.sh","postgres:5432", "--", "/start.sh"]
```