

IIC 2143 – Ingeniería de Software

# Arquitectura de Software

M. Trinidad Vargas  
mtvargas1@uc.cl

# Arquitectura de software

“La arquitectura de software de un sistema representa las decisiones de diseño relacionadas con la estructura y el comportamiento general del sistema. La arquitectura ayuda a las partes interesadas a comprender y analizar cómo el sistema logrará cualidades esenciales como la modificabilidad, la disponibilidad y la seguridad.”

Software Architecture in Practice”,  
Len Bass, Paul C. Clements, Rick Kazman.

# Arquitectura de software

“Architecture is about the important stuff. Whatever that is”  
Ralph Johnson



# Arquitectura de Software

Los aspectos más importantes incluyen

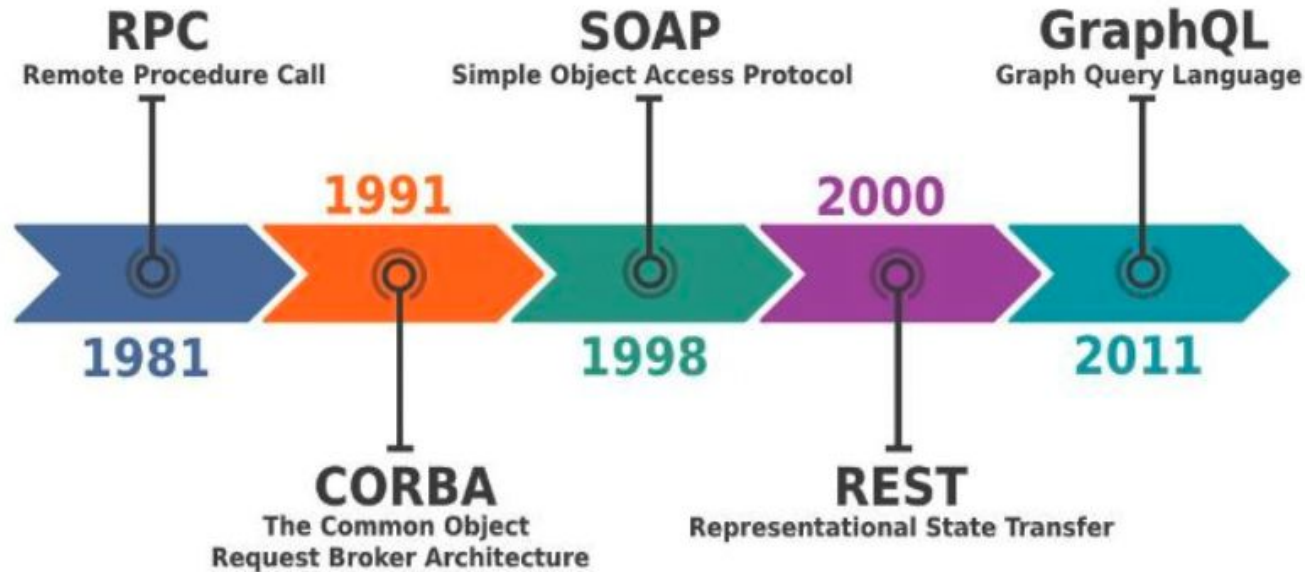
- Centrarse en la estructura
- Anticiparse a decisiones costosas
- Hacer explícitas las decisiones para tener una buena calidad

# Requerimientos no funcionales

En el Siglo 21 se esperan ciertos requerimientos no funcionales para el software:

- **Maintainability:** Sea desarrollado y mantenido por muchos años
- **Scalability:** Soporte a millones de usuarios
- **Reliability:** Disponible 24/7
- **Efficiency:** Que tenga buena latencia

# Un poco de historia



# Patrones arquitectónicos

Los patrones arquitectónicos son soluciones a ciertos problemas en un contexto. Estos patrones suelen usarse en la práctica y demostraron tener beneficios.

Algunos son:

- Arquitectura Cliente–Servidor o Multi–Tier
- Arquitectura por capas
- Arquitectura orientada a servicios
  - Arquitectura orientada a servicios clásica (SOA)
  - Arquitectura orientada a microservicios

# Arquitectura Cliente - Servidor

## Cliente

Un cliente puede considerarse una máquina o un programa que tiene la capacidad y una forma de enviar solicitudes (request) a través de internet.

- No necesariamente es un browser.
- Un computador puede tener varios clientes.



POSTMAN



# Arquitectura Cliente - Servidor

## Servidor

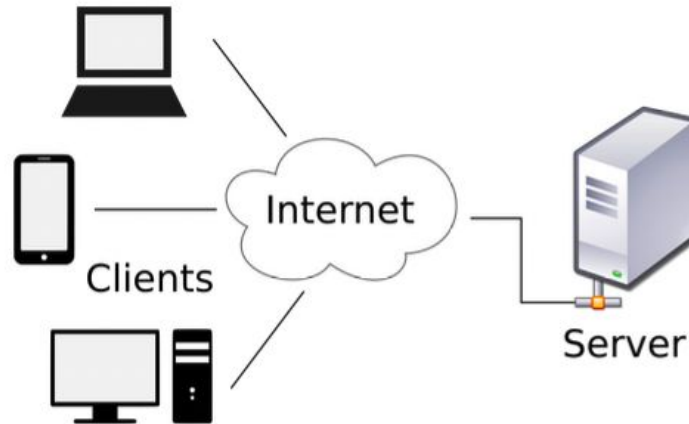
Un servidor no es necesariamente un dispositivo (i.e. computadora).

- Algunas veces las computadoras de alto rendimiento son llamadas servidores porque ejecutan programas que dan servicios.
- Un servidor puede atender múltiples clientes al mismo tiempo.

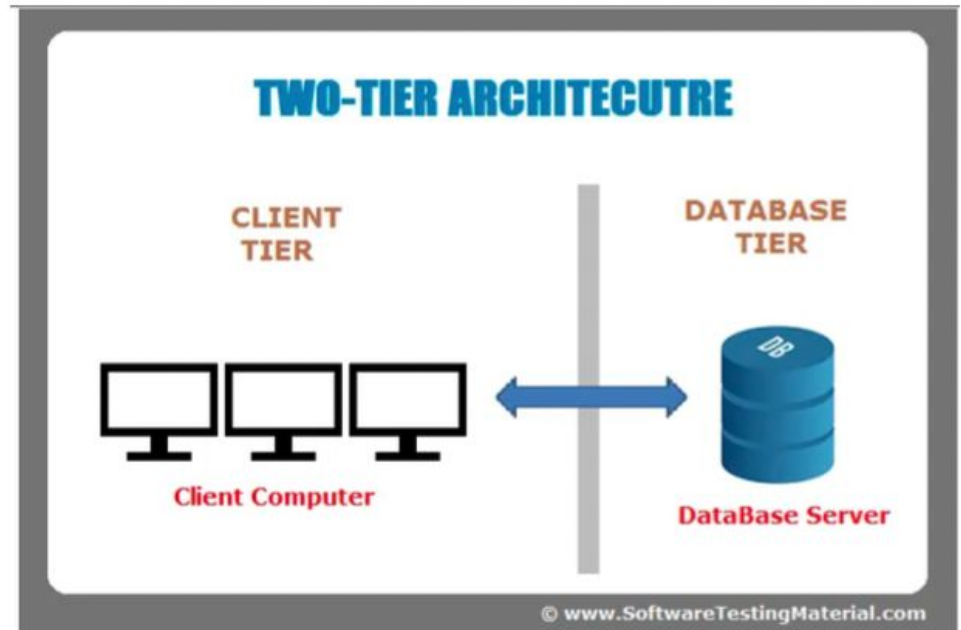
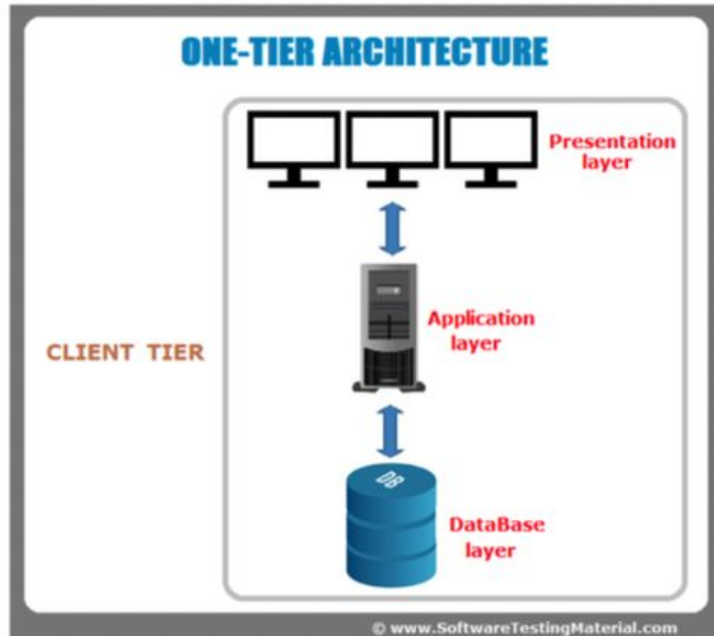


# Arquitectura Cliente - Servidor

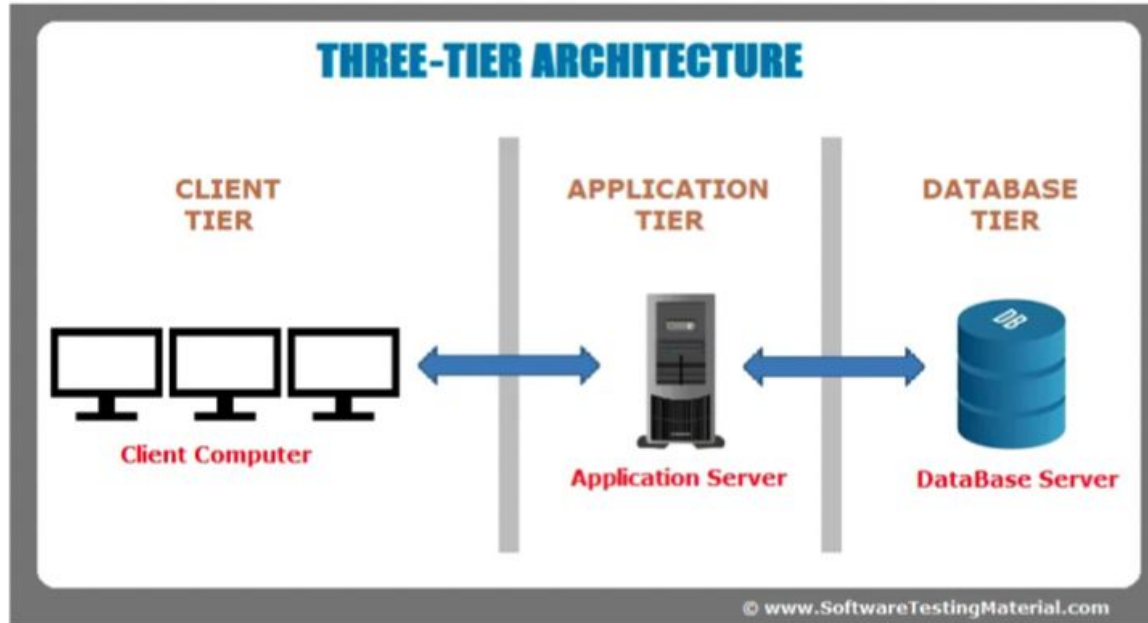
Es una arquitectura de software donde la responsabilidad se divide entre el cliente y el servidor (client-server) o en más de dos (tiered).



# Arquitectura Cliente - Servidor



# Arquitectura Multi-Tier



# Arquitectura Cliente - Servidor o Multi-Tier

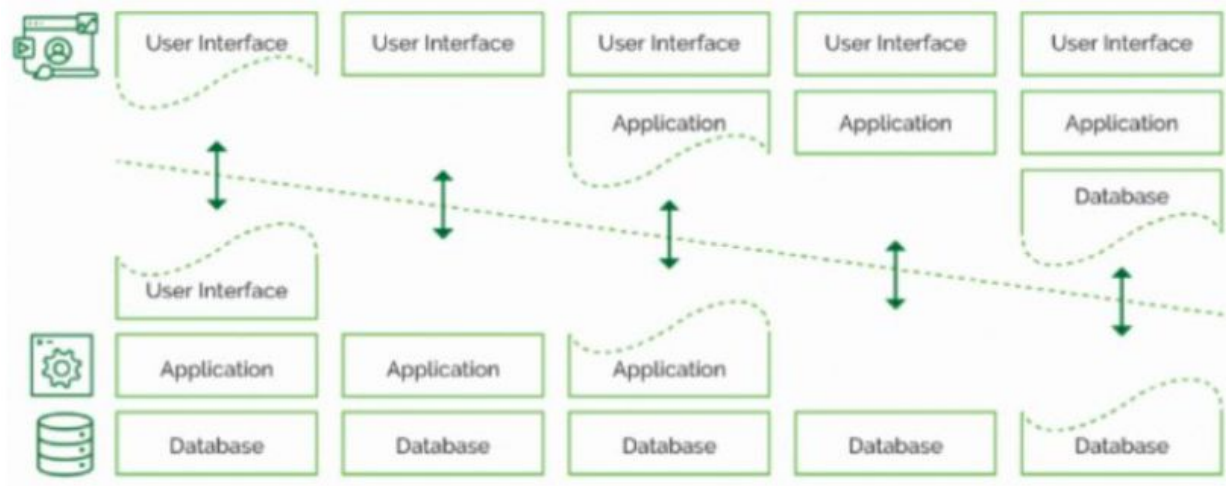
La arquitectura **client-server** era popular en los 80's y 90's porque facilitaba escalar el número de usuarios y permitir el uso de interfaces gráficas sin sobrecargar.

La arquitectura **Three-Tier** se generó como resultado de la evolución de la arquitectura client-server. En Three-Tier usualmente el cliente es responsable sólo de la interfaz del usuario.

# Arquitectura Cliente - Servidor o Multi-Tier

En las arquitecturas multi-tier es posible realizar variaciones en las responsabilidades que toma el cliente.

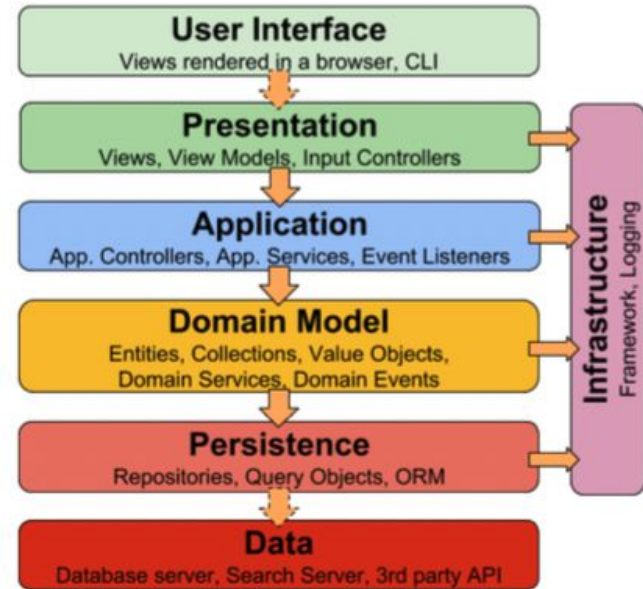
A lo largo del tiempo ha cambiado la popularidad de cada una de estas opciones.



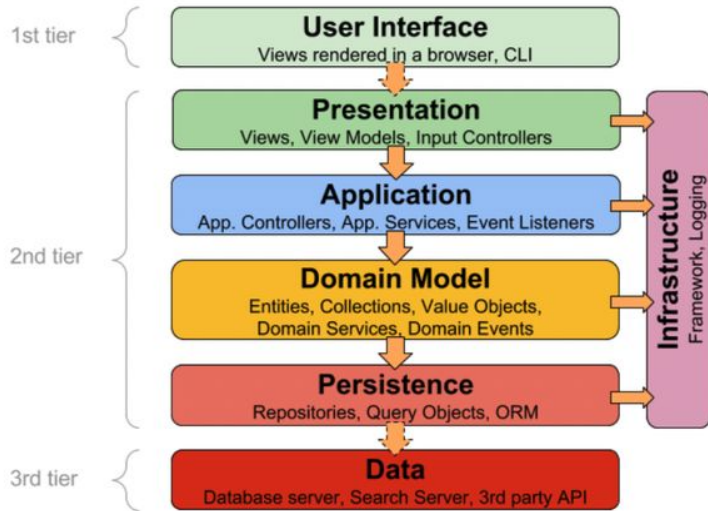
# Arquitectura por Capas (Layered Arch.)

La arquitectura por capas organiza el sistema en capas con funcionalidad relacionada con cada capa.

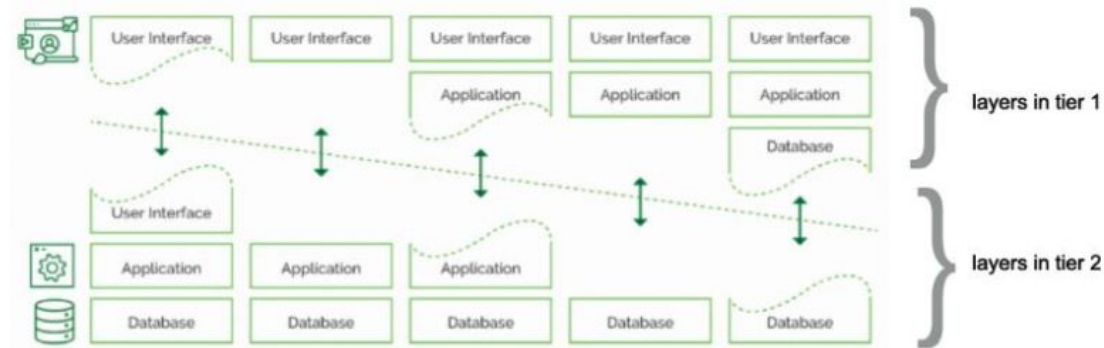
Una capa brinda servicios a la capa de encima, de modo que las capas de nivel inferior representan servicios núcleo que es probable se utilicen a lo largo de todo el sistema.



# Combinación de Capas y Multi-Tier





[www.herbertograca.com](http://www.herbertograca.com)





# Escalabilidad

¿Nuestra arquitectura nos permite escalar?

- 100 usuarios: funcionando 
- 1000 usuarios: funcionando 
- 1000000 usuarios: no necesariamente

# Escalabilidad

- Vertical scaling: Agregamos más recursos (CPU, RAM, etc).
- Horizontal scaling: Agregamos más servidores.



Vertical Scaling  
(Scaling up)

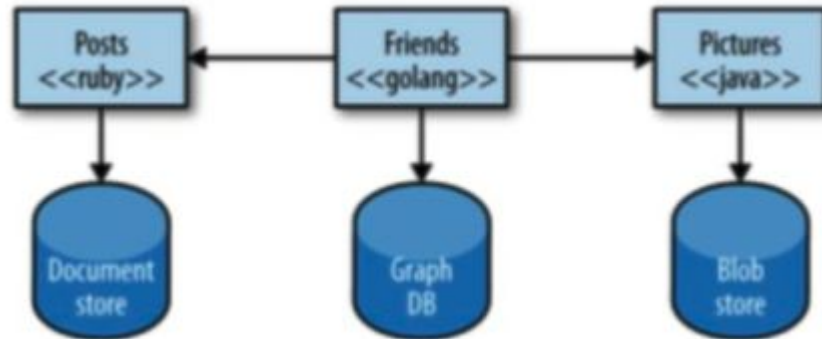


Horizontal Scaling  
(Scaling out)

# Arquitectura orientada a servicios

En las arquitecturas clásicas un componente se implementa como un módulo, un paquete o una librería compartida que debe ser ensamblado con el todo.

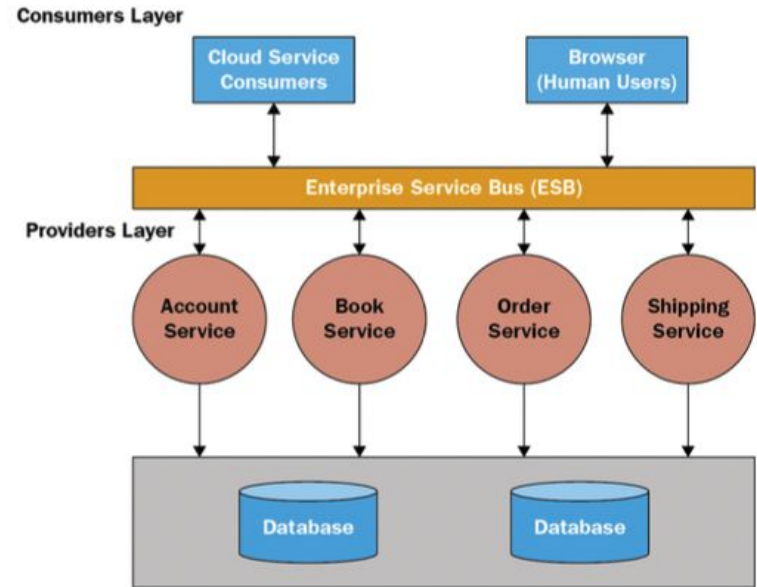
Estos servicios se comunican entre ellos usando protocolos. Se dice que en este caso las componentes están débilmente acopladas.



# Arquitectura orientada a servicios

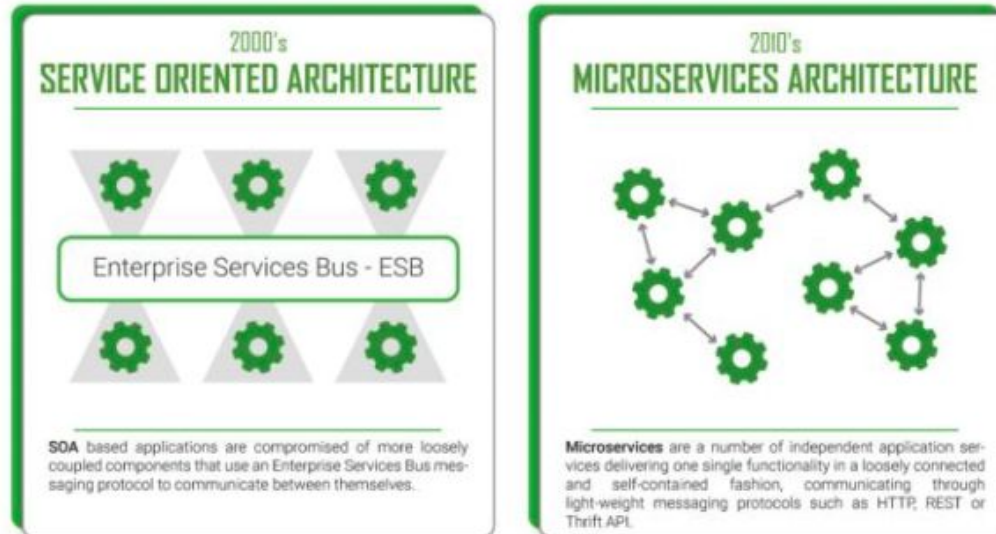
Los componentes del sistema son servicios. Por lo tanto, el sistema es una combinación de servicios.

Usualmente, los servicios interactúan mediante una unidad *enterprise service bus* que se encarga de combinarlos adecuadamente y proveer elementos necesarios para realizar transacciones complejas.



# Arquitectura orientada a microservicios

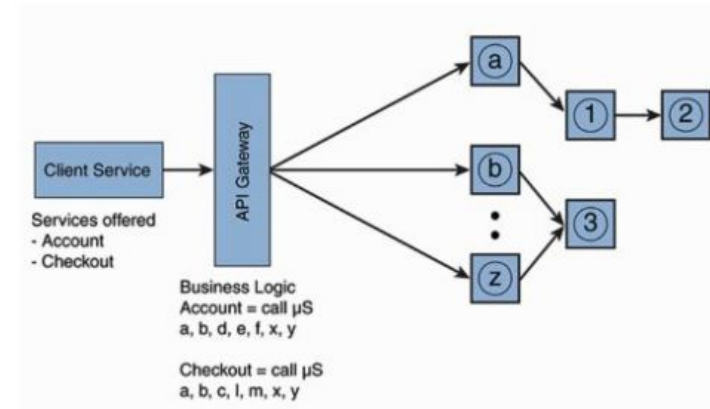
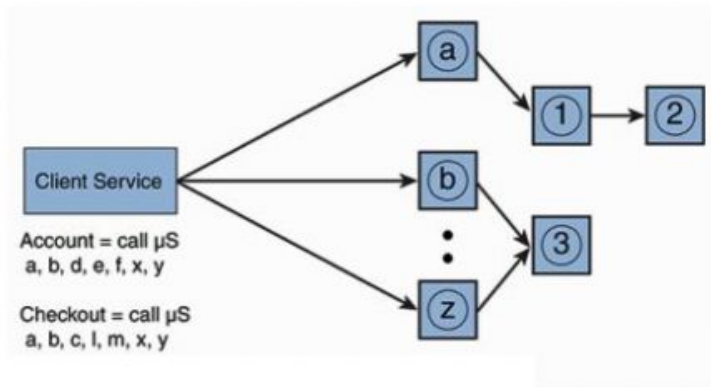
En esta arquitectura el software está compuesto por pequeños servicios independientes que se comunican a través de protocolos bien definidos.



# Arquitectura orientada a microservicios

Un API Layer o API Gateway actúa como una fachada de los servicios hacia el lado del cliente y que representa lo que realmente queda expuesto de los servicios.

- Simplifica el uso de los servicios para el cliente
- Permite cambiar los protocolos de comunicación



# Arquitectura orientada a microservicios

## Ventajas

- Desarrollo independiente.
- Escalabilidad más sencilla.
- Descentralización de datos.
- Flexibilidad tecnológica.

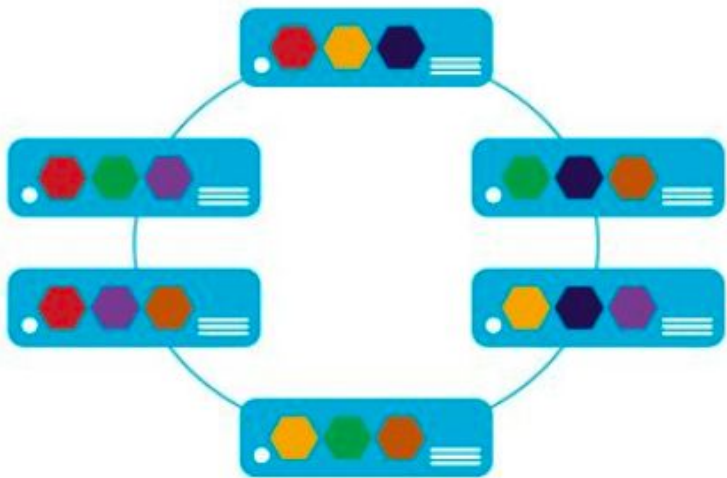
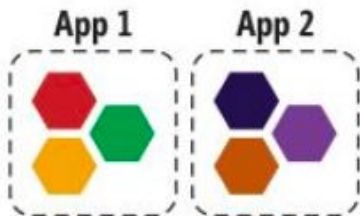
## Desventajas

- Puede haber pérdida de desempeño general.
- Es más difícil de asimilar y entender.
- El versionamiento es más complejo.
- Mayor esfuerzo de desarrollo.
- Es difícil asegurar transacciones o consistencia.

## Microservices Approach

A microservice approach segregates functionality into small autonomous services.

And scales out by **deploying independently** and replicating these services across servers/VMs/containers.



## VS. Traditional Approach

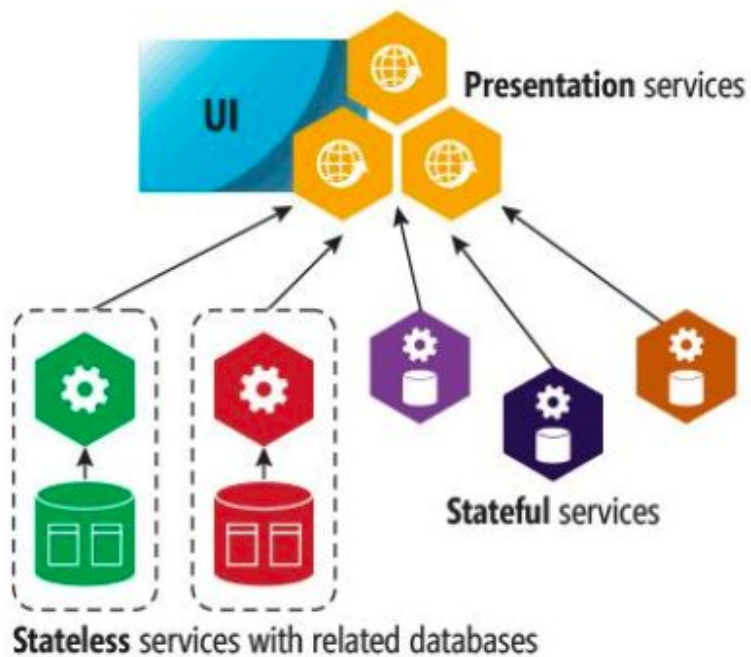
A traditional application (Web app or large service) usually has most of its functionality within a single process (usually internally layered, though).

And scales by cloning the whole app on multiple servers/VMs/containers.





## Microservices Approach



Model/Database per Microservice

## Traditional Application

- Single app process or 3-Tier approach
- Several modules
- Layered modules

### 3-Tier Approach

### Single App Process



Or



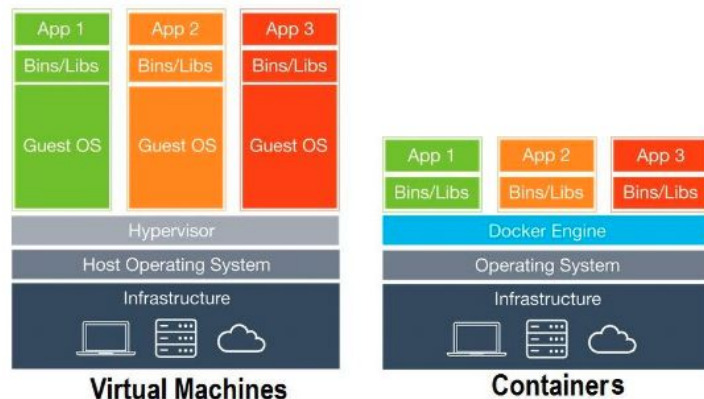
Single Monolithic Database

# Plataformas y herramientas

## Docker

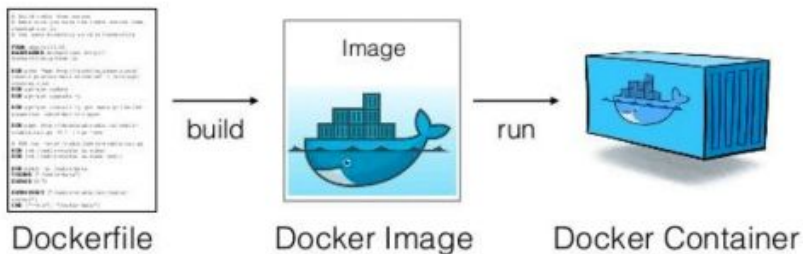
Docker es una plataforma abierta para desarrollar, enviar y ejecutar aplicaciones.

Permite separar las aplicaciones de la infraestructura para que pueda entregar software rápidamente. Con Docker, puede administrar la infraestructura de la misma manera que administra las aplicaciones.



# Docker

El Dockerfile definir instrucciones paso a paso para construir una imagen de contenedor reproducible con todo lo necesario para ejecutar una aplicación.



```
# Especificamos la imagen de la que se heredará

FROM python:3.6-alpine

# Definimos algunas variables de ambiente

ENV LIBRARY_PATH=/lib:/usr/lib

ENV PYTHONUNBUFFERED 1

# Ejecutamos algunos comandos para aprovisionar la imagen

RUN mkdir /code

WORKDIR /code

ADD requirements.txt /code/

RUN pip install -r requirements.txt

# Copiamos el código

ADD . /code/

# Exponemos el puerto por donde se interactuará con la aplicación

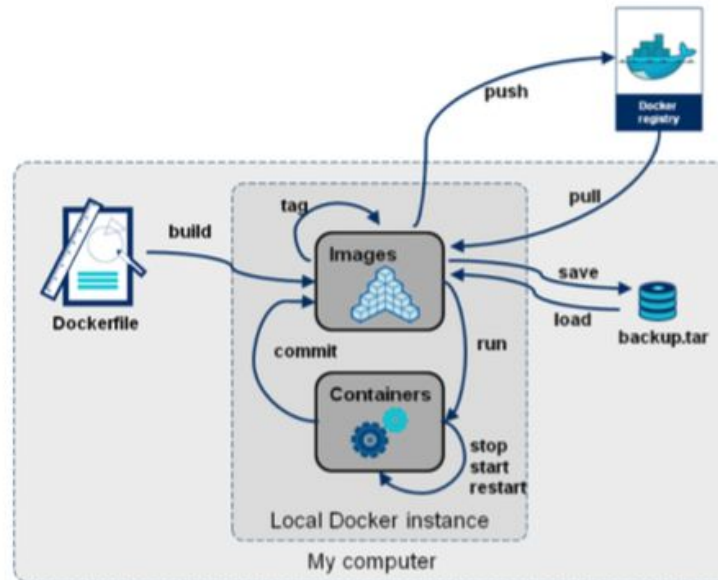
EXPOSE 8000

# Definimos el comando de entrada a la aplicación

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

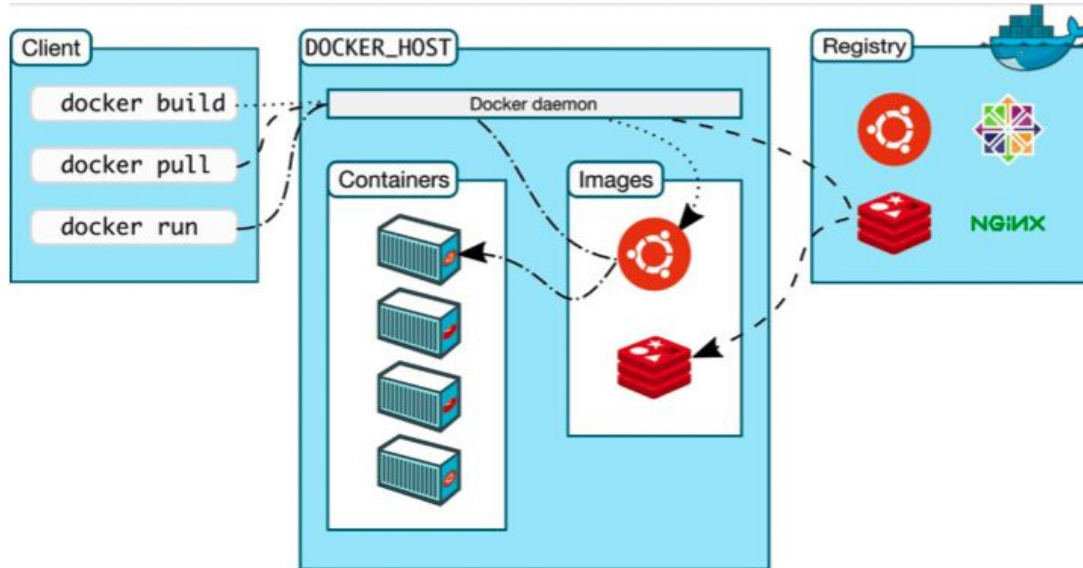
# Docker

Permite diseñar aplicaciones como un conjunto de contenedores que deben funcionar juntos con recursos compartidos y canales de comunicación adecuados.



# Docker

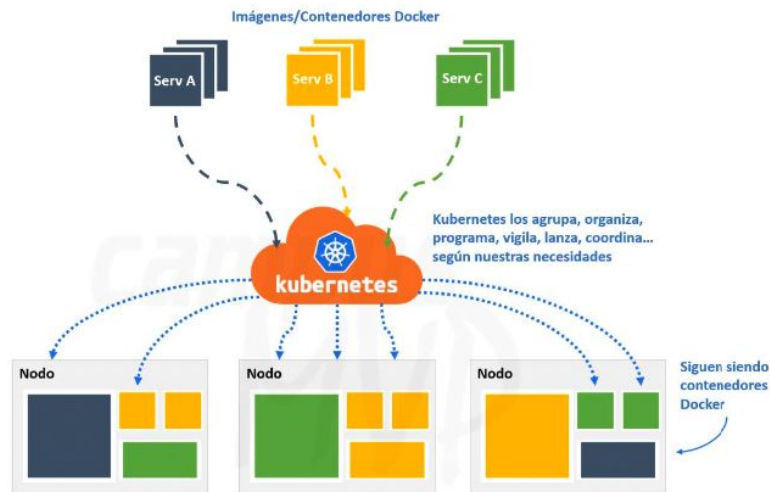
Además permite subir imágenes a la nube a partir de Docker Registry.



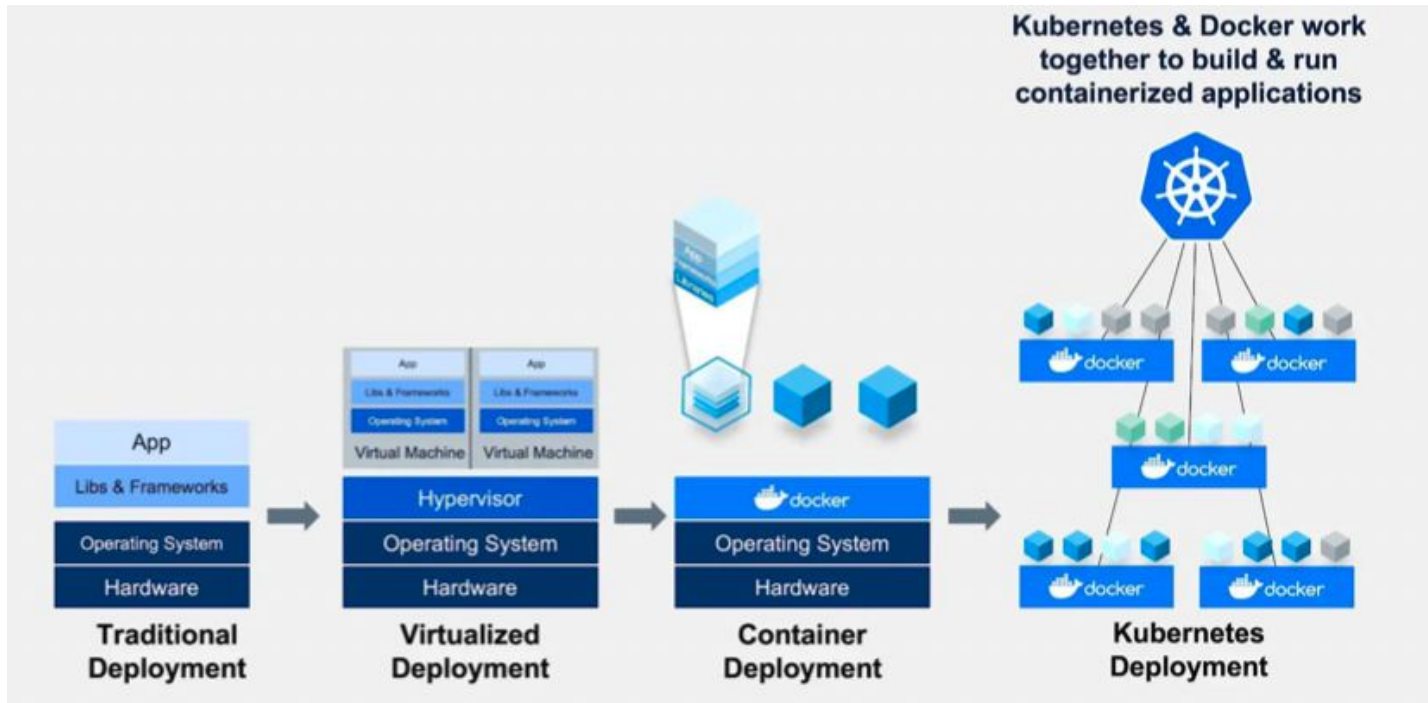
# Kubernetes

Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios.

Facilita la automatización y la configuración declarativa.

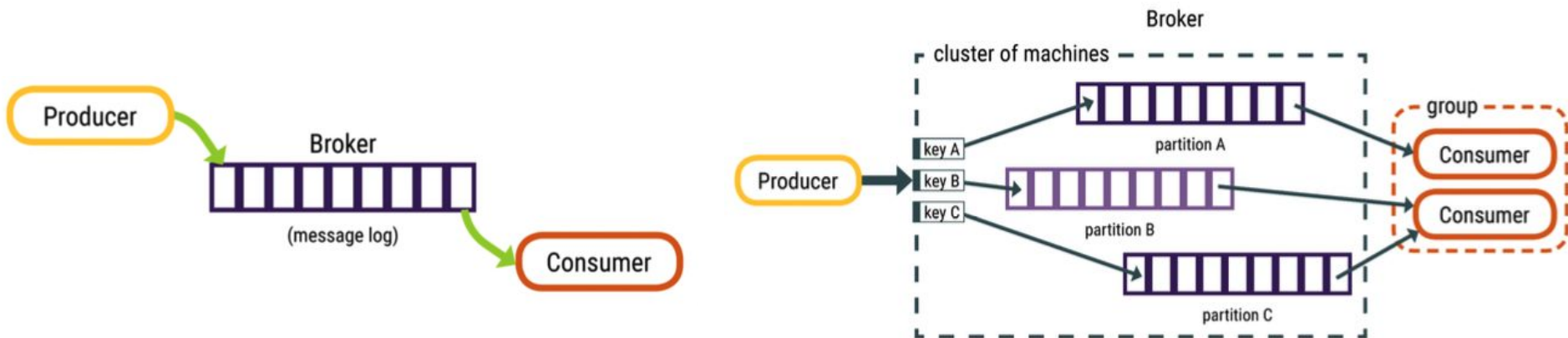


# Historia de Kubernetes



# Kafka

Kafka funciona como un sistema de mensajería distribuido, donde un Producer envía eventos a un Broker (Kafka), y los Consumers los leen de forma asíncrona, permitiendo desacoplar los componentes del sistema.





# Material adicional

- Documentación Docker: <https://docs.docker.com/>
- Documentación Kubernetes: <https://kubernetes.io/docs/home/>
- Documentación Kafka: <https://kafka.apache.org/>