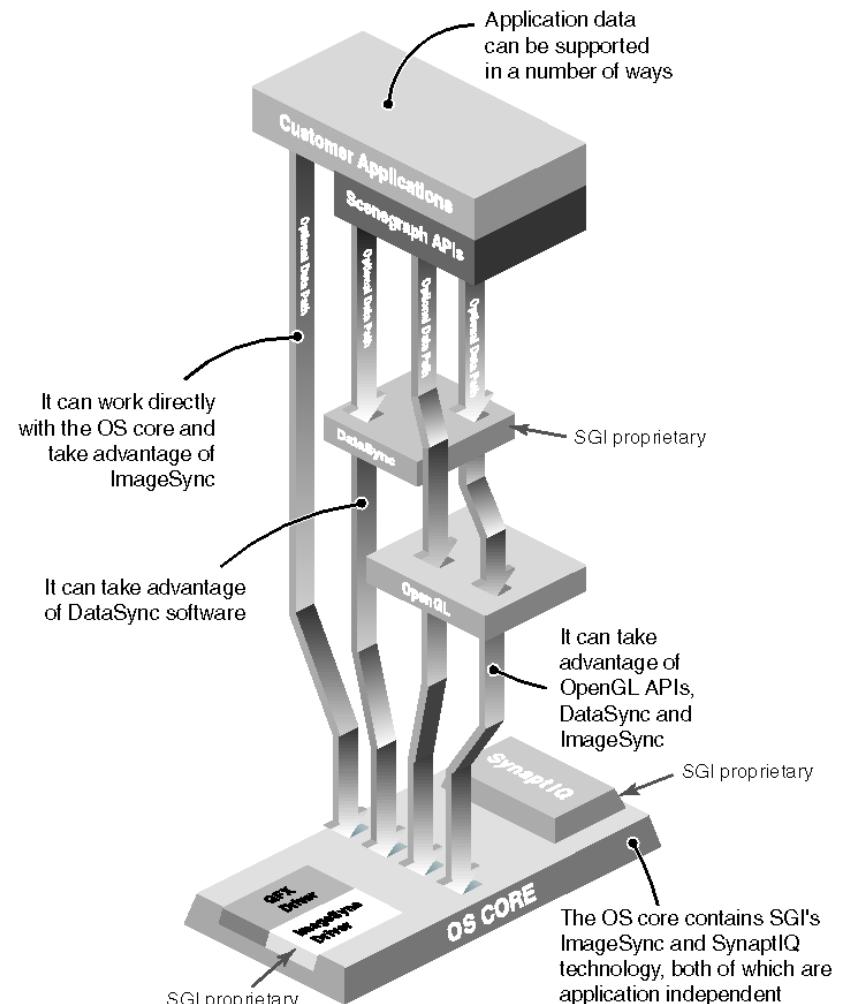


# Arquitectura de Software

¿De qué estamos hablando ?

- estructura: descomposición en componentes y sus interacciones
- grandes decisiones de diseño basadas en requerimientos (incluyendo los no funcionales)



# Factores que influencian las decisiones de arquitectura

- ▶ performance - minimizar el número de comunicaciones
- ▶ security - organización en capas
- ▶ availability - redundancia controlada
- ▶ maintainability - componentes autocontenidoas fácilmente intercambiables

# Arquitectura vs Diseño

- ▶ La separación no es tan nítida pero hay aspectos que son claramente de arquitectura
  - ▶ la "forma" del sistema: client-server, web-based, native mobile client, etc
  - ▶ estructura del software (componentes, capas)
  - ▶ tecnologías (lenguaje, plataforma de despliegue)
  - ▶ frameworks
  - ▶ enfoque de logro de performance, escalabilidad, etc

# Arquitectura y Agilidad

- ▶ Una buena arquitectura habilita la agilidad
  - ▶ por ejemplo arquitectura de microservicios (mas adelante) facilita desarrollo incremental
  - ▶ separación clara de componentes facilita los tests
  - ▶ se puede agregar funcionalidades con mayor facilidad sobre arquitectura sólida

# Architectural Drivers

- ▶ requerimientos funcionales
- ▶ atributos de calidad (performance, seguridad, etc)
- ▶ restricciones
- ▶ principios (consistencia y claridad del código)

# Atributos de Calidad

- ▶ performance (tiempo de respuesta, latencia, etc)
- ▶ escalabilidad
- ▶ disponibilidad
- ▶ seguridad
- ▶ continuidad operacional (recuperación)
- ▶ accesibilidad (usuarios con necesidades)
- ▶ monitoreo

# Algunas Restricciones

- ▶ Tiempo y presupuesto (las obvias)
- ▶ Tecnológicas
  - ▶ lista de tecnologías aprobadas
  - ▶ sistemas existentes e interoperabilidad
  - ▶ plataforma destino
  - ▶ open source (yes/no)
  - ▶ madurez de tecnologías a incluir
  - ▶ relaciones con los proveedores
  - ▶ fallas anteriores
- ▶ Relativas a personas
  - ▶ tamaño del equipo
  - ▶ habilidades del equipo
  - ▶ se pueden agregar especialistas
  - ▶ la mantención la hará el mismo equipo ?

# Principios

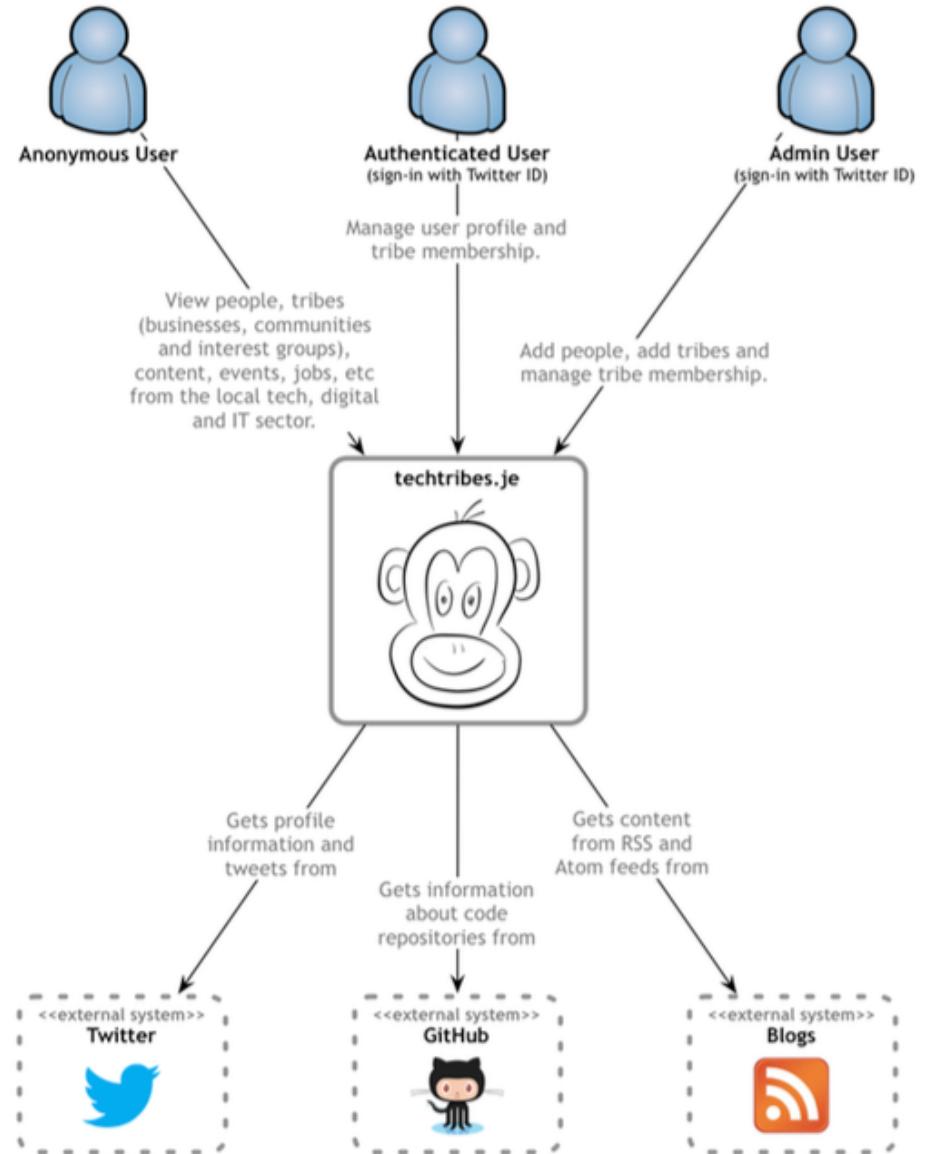
- ▶ De Desarrollo
  - ▶ estandares de codificación
  - ▶ full unit testing
  - ▶ revisión de código
- ▶ Arquitectónicos
  - ▶ separación de lógica de negocios
  - ▶ componentes sin estado
  - ▶ consistencia eventual
  - ▶ procedimientos almacenados
  - ▶ gestión del sistema
  - ▶ auditoría flexibilidad
  - ▶ extensibilidad
  - ▶ mantenibilidad
  - ▶ aspectos legales y regulatorios
  - ▶ localización e internacionalización

# Cómo describir la Arquitectura

- ▶ Son fundamentales los diagramas
- ▶ Puede usarse UML (despliegue, componentes)
- ▶ Generalmente se usan diagramas más simples
- ▶ Descripción en 3 niveles: contexto, contenedores, componentes

# Contexto

- ▶ Quien o quienes lo usarán
- ▶ De que se trata lo que estamos construyendo
- ▶ Como encaja en la infraestructura existente

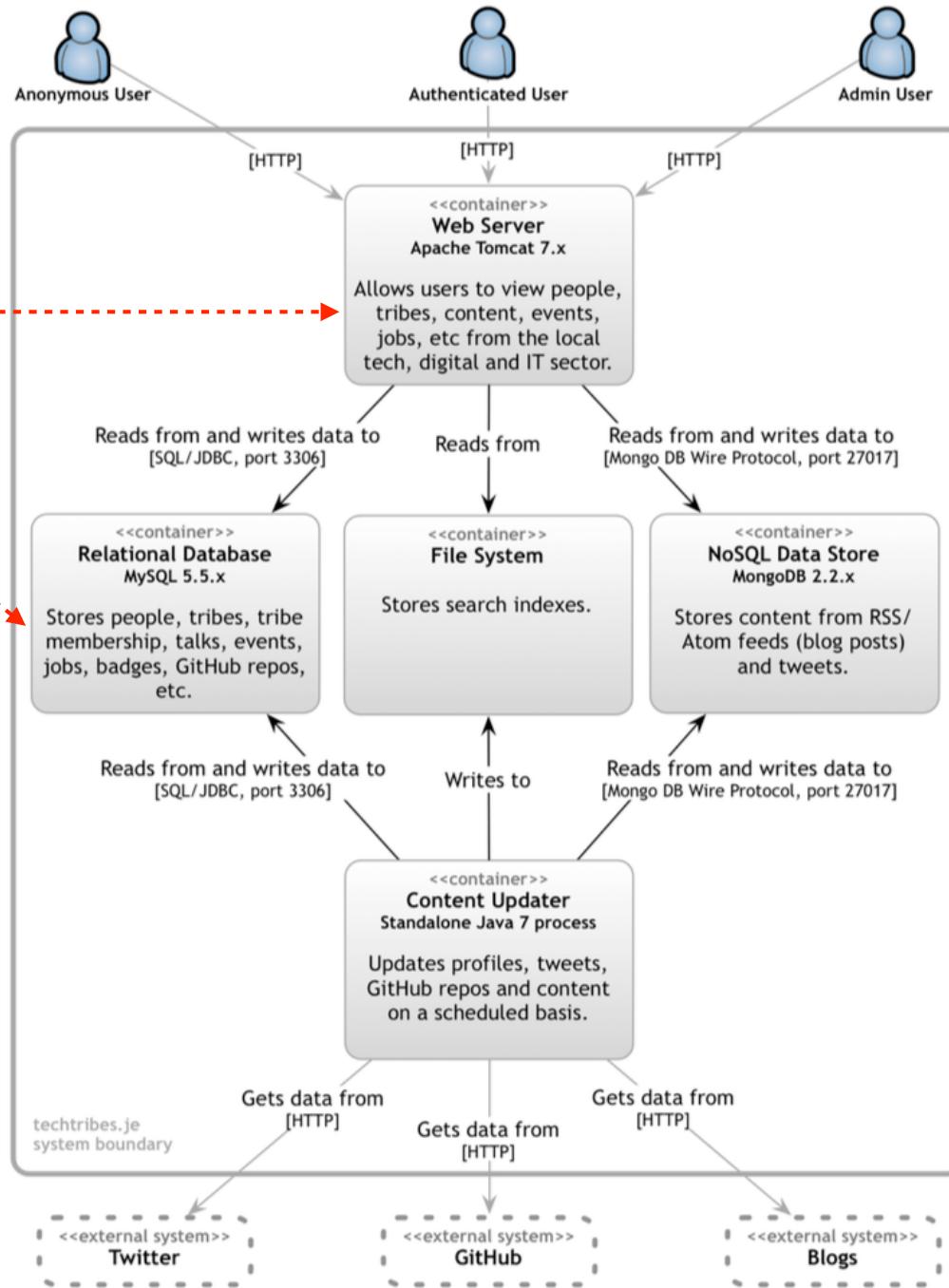


# Containers: los grandes subsistemas

- ▶ Forma general del sistema
- ▶ decisiones tecnológicas de alto nivel
- ▶ distribución de responsabilidades
- ▶ comunicación entre containers
- ▶ donde se debe desarrollar qué código

# Ejemplo

Containers



# Componentes

- ▶ ¿De qué componentes o servicios se compone el sistema?
- ▶ corresponde a un servicio en arquitectura SOA o de microservicios
- ▶ en que contenedor se ubica

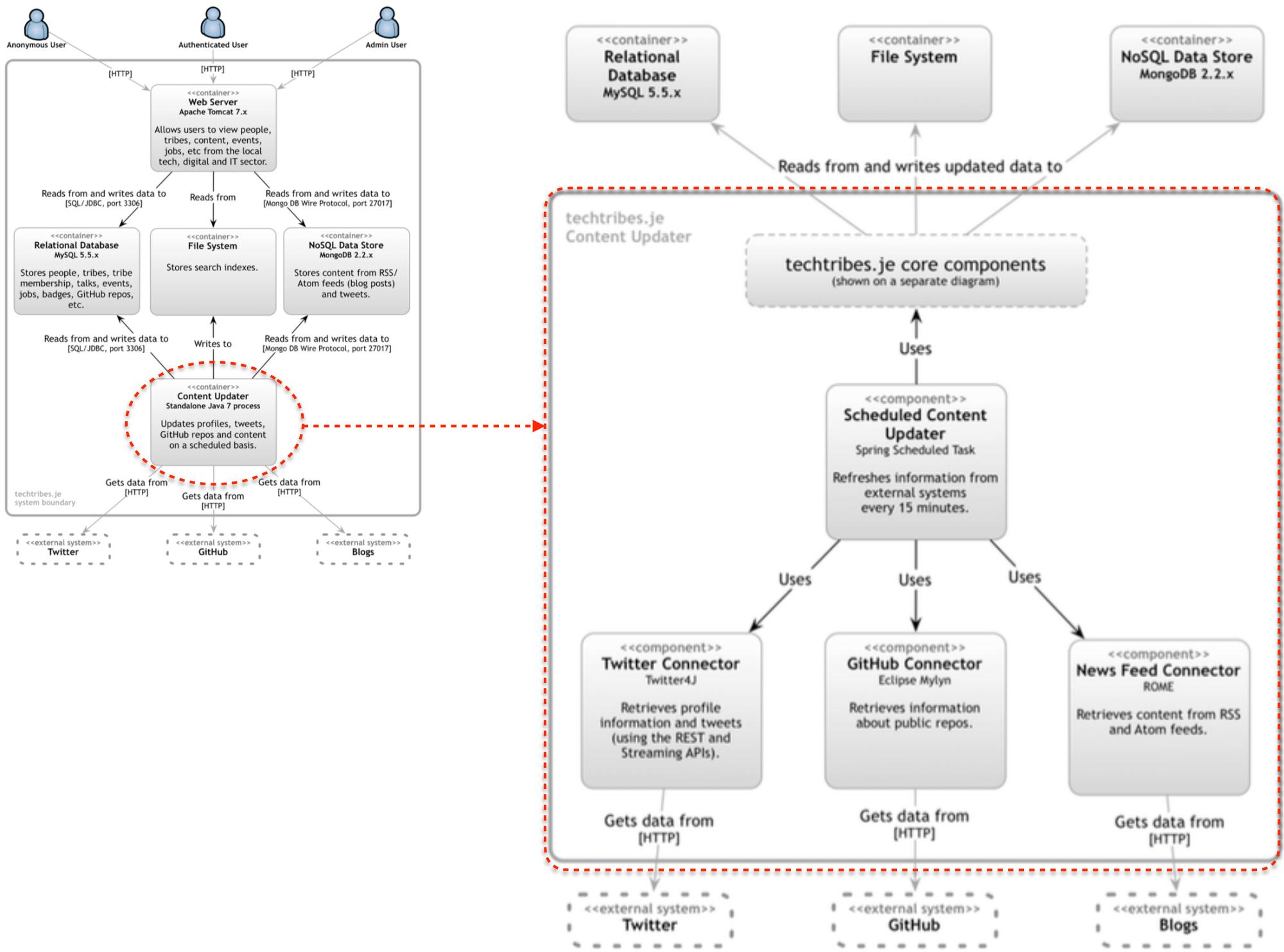


Container

Component

# Ejemplo para Sistema de Riesgo Financiero

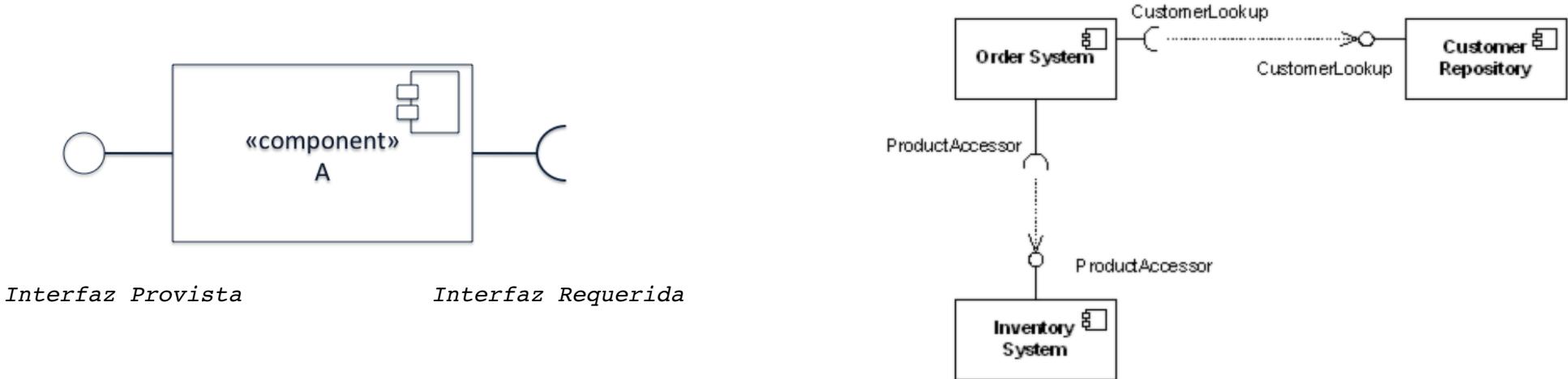
- ▶ Importador de datos de mercado
- ▶ Calculador de riesgos
- ▶ Servicio de autenticación
- ▶ Auditoría
- ▶ Servicio de Notificaciones
- ▶ Monitoreo



# UML para Arquitectura

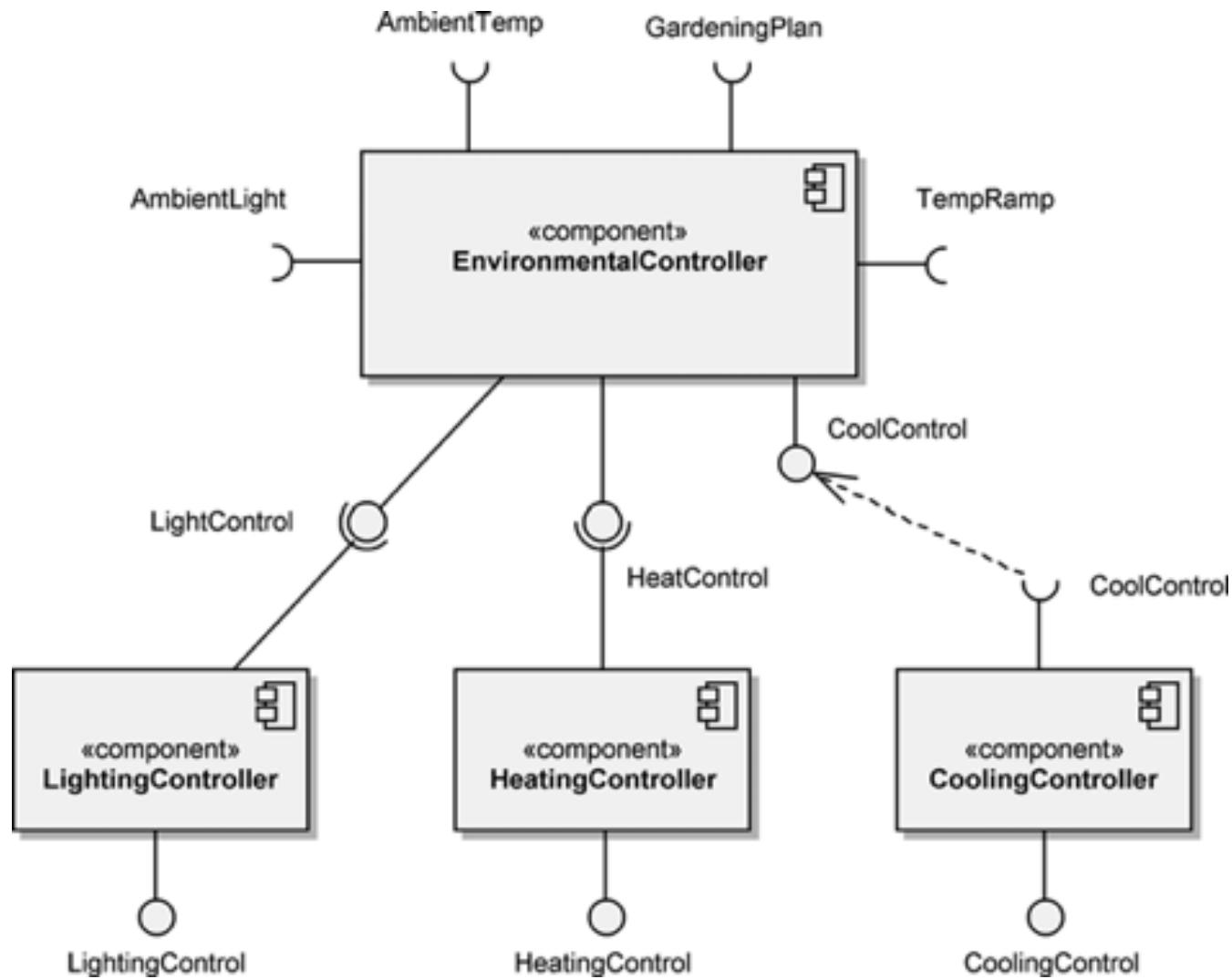
- ▶ Diagrama de paquetes
- ▶ Diagrama de componentes
- ▶ Diagrama de despliegue

# Diagrama UML de Componentes



- ▶ Un componente representa una parte modular de un sistema, que encapsula su contenido y cuya manifestación (el o los artefactos físicos correspondientes) es reemplazable dentro de su entorno.
- ▶ Un componente define su comportamiento en términos de interfaces provistas e interfaces requeridas

# Diagrama UML de Componentes

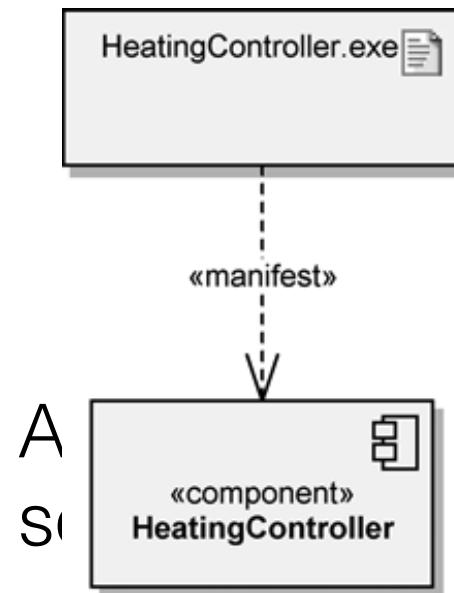


# Diagrama de Despliegue (deployment)

- ▶ Muestra la asignación de artefactos de software concretos (archivos fuentes, archivos ejecutables, scripts y tablas de una base de datos) a nodos computacionales (algo que ofrece servicios de procesamiento).
- ▶ Es útil para comunicar la arquitectura física o de despliegue (instalación).

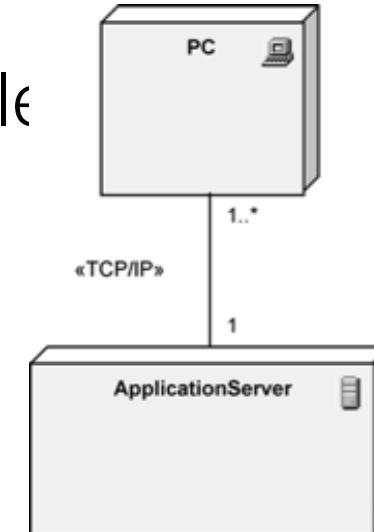
# Artefactos y Nodos

Nodo: recurso computacional

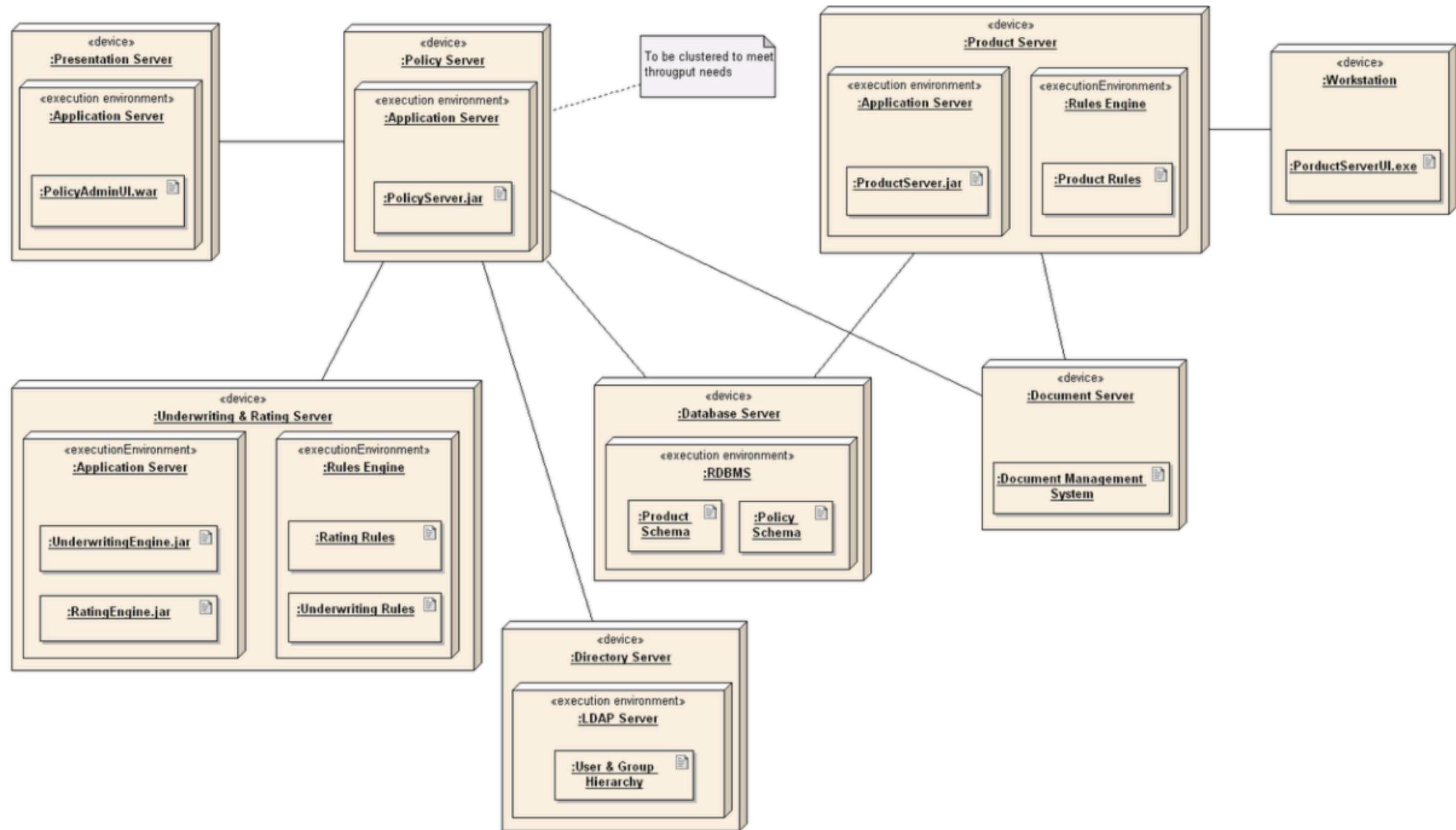


device (computer, modem, etc)  
execution environment (oracle  
engine, J2EE server)

Artefactos: componentes físicos que implementan el arte del



# Diagrama UML de Despliegue



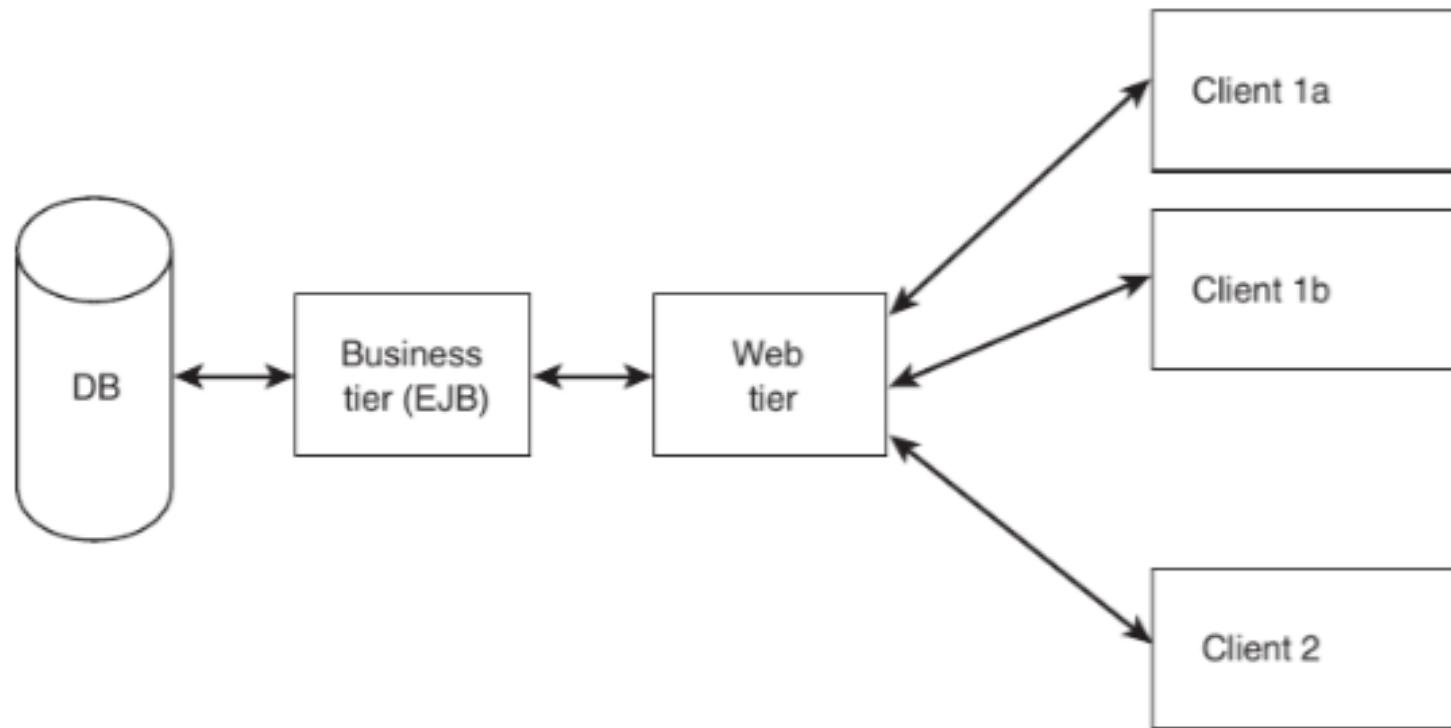
# Patrones Arquitectónicos

- ▶ Similar a patrones de diseño
- ▶ Soluciones que se encuentran a menudo en el mundo real
- ▶ Permiten no tener que partir de cero

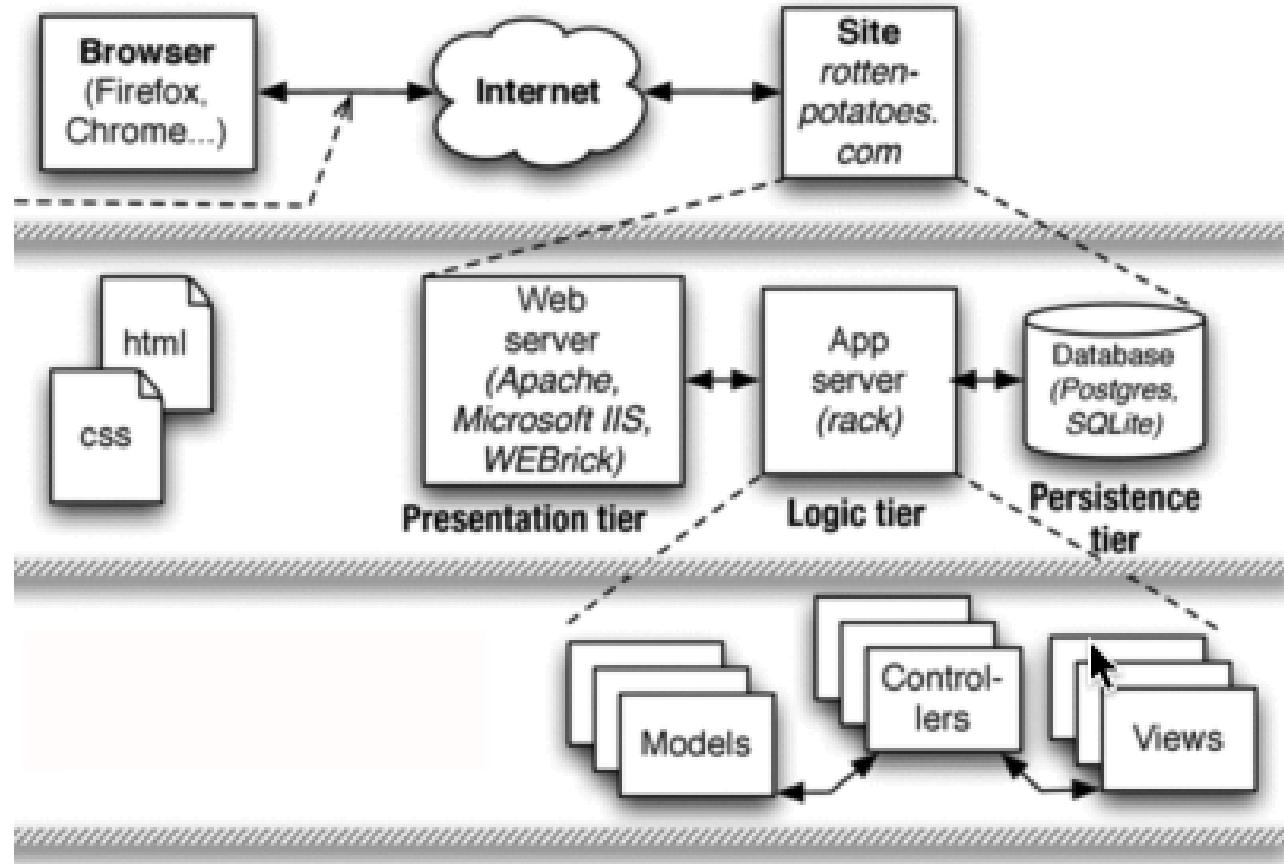
# Client-Server y Tiered

- ▶ Dos grandes roles: cliente y servidor (2 Tier)
- ▶ Servidor recibe y procesa solicitudes de cliente
- ▶ Tremendamente popular en los 80s y 90s dio origen a una variación conocida como “3 capas” o “thin client” (3 Tier)
- ▶ Thin client
  - ▶ tier 1: server
  - ▶ tier 2: application server
  - ▶ tier 3: client

# Arquitectura de referencia J2EE

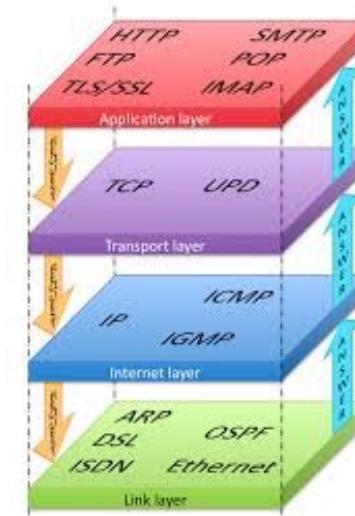
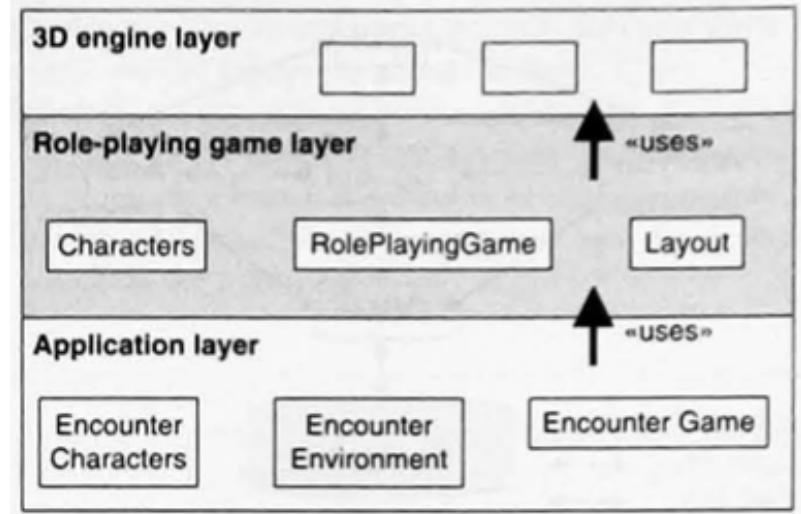


# Aplicación Web (Ruby)



# Arquitectura de Capas

- ▶ una capa es una colección (lógica) coherente de componentes
- ▶ usa y es usada por a lo más una capa

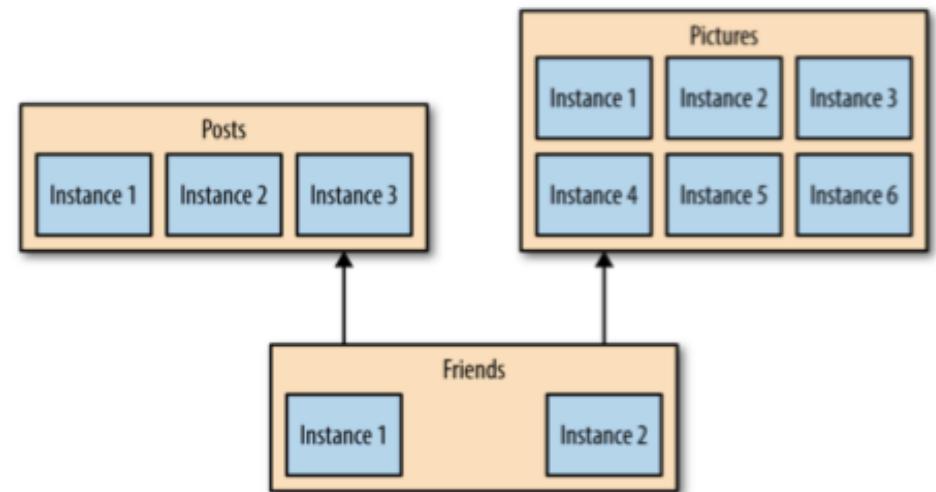
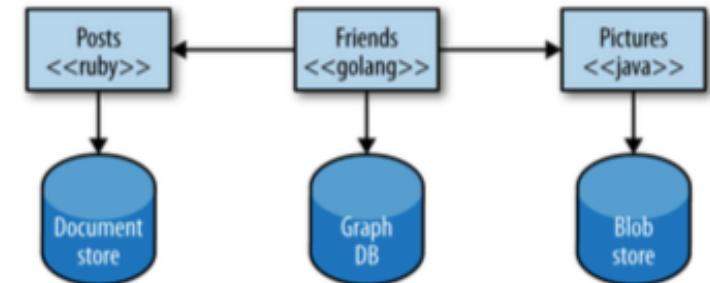


# Arquitectura de Servicios: componentes débilmente acopladas

- ▶ una componente es una unidad de software que puede ser reemplazada o mejorada (upgrade) en forma independiente
- ▶ tradicionalmente las componentes se presentan en forma de módulos o librerías compartidas
- ▶ un servicio es una componente que no corre en el mismo proceso y se comunica con el resto mediante un protocolo como http o rpc

# Ventajas

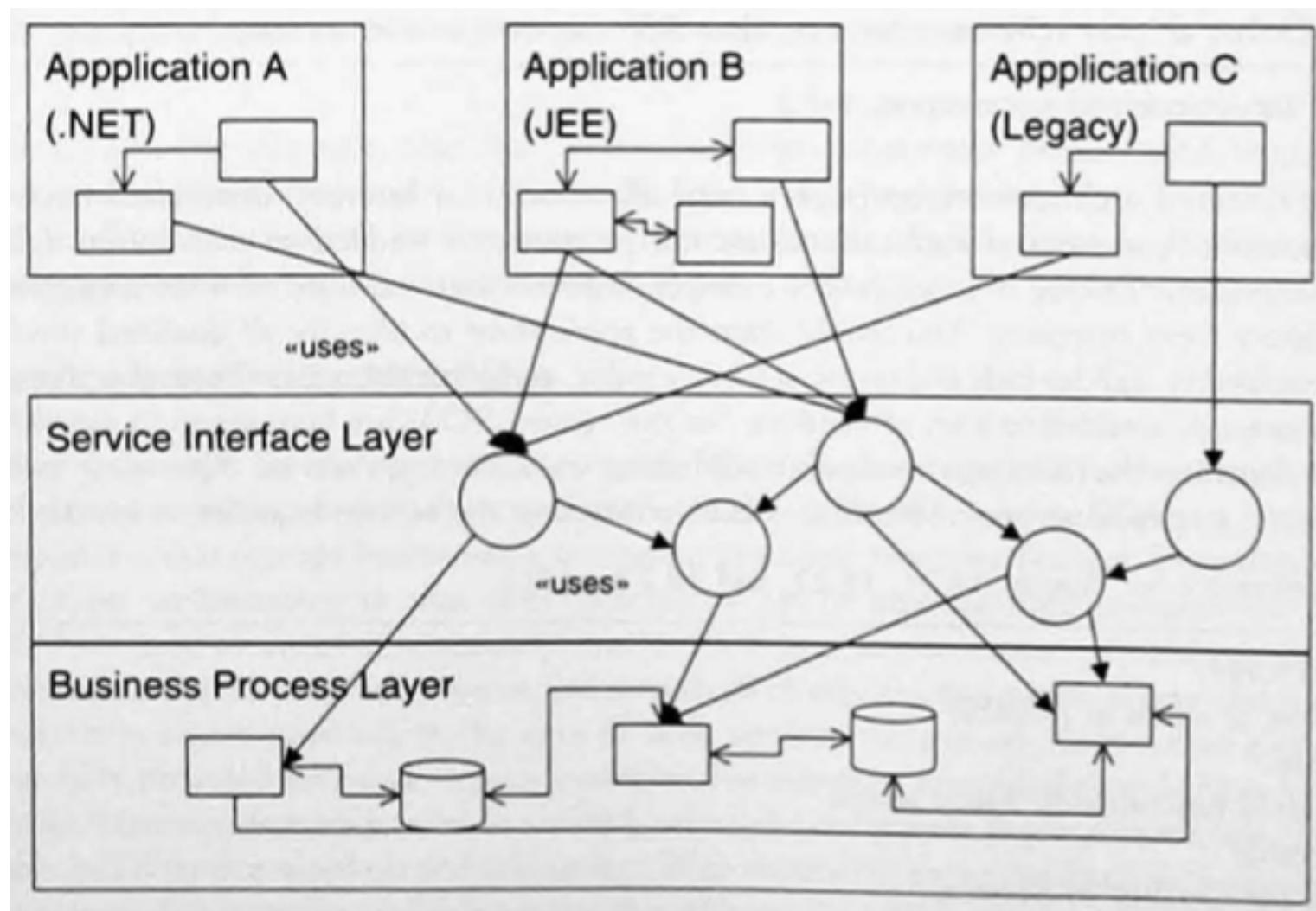
- ▶ deployable en forma independiente
  - ▶ si se hace un cambio de una componente no es necesario redeployar la aplicación
- ▶ interfaz explícita
  - ▶ hace más difícil violar acuerdos
- ▶ heterogeneidad tecnológica
- ▶ resiliencia
- ▶ escalamiento flexible



# Service Oriented Architecture (SOA)

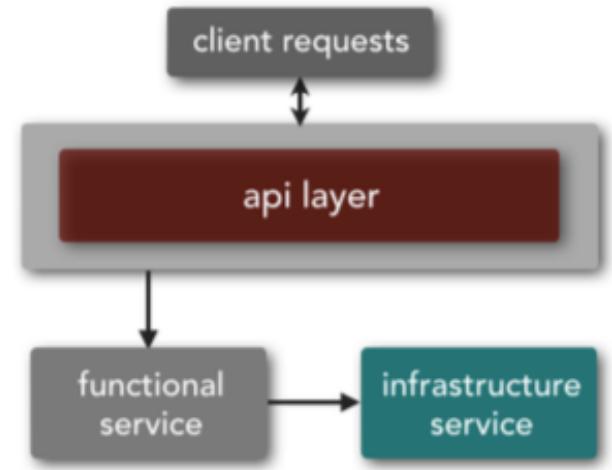
- ▶ mediados de los 2000
- ▶ Sistema se concibe como una combinación de servicios
- ▶ Servicios proveen funcionalidades de acuerdo a una especificación de interfaz
- ▶ Pueden ser combinados en forma dinámica
- ▶ Intimamente ligado a la popularidad de los Web Services y a la formalización de los procesos de negocio

# SOA



# La llegada de los microservicios

- ▶ 2011 - 2012
- ▶ algunos se refieren a "SOA hecho bien"
- ▶ primera generación de SOA comenzó a hacerse demasiado compleja
  - ▶ SOAP -> middleware
- ▶ representa una maduración de las ideas de SOA a la luz de la experiencia práctica
- ▶ no requiere capa de middleware
  - ▶ cada servicio expone una API (contrato)
- ▶ la mayor parte son servicios funcionales con algunos servicios de infraestructura (no se exponen al mundo externo)



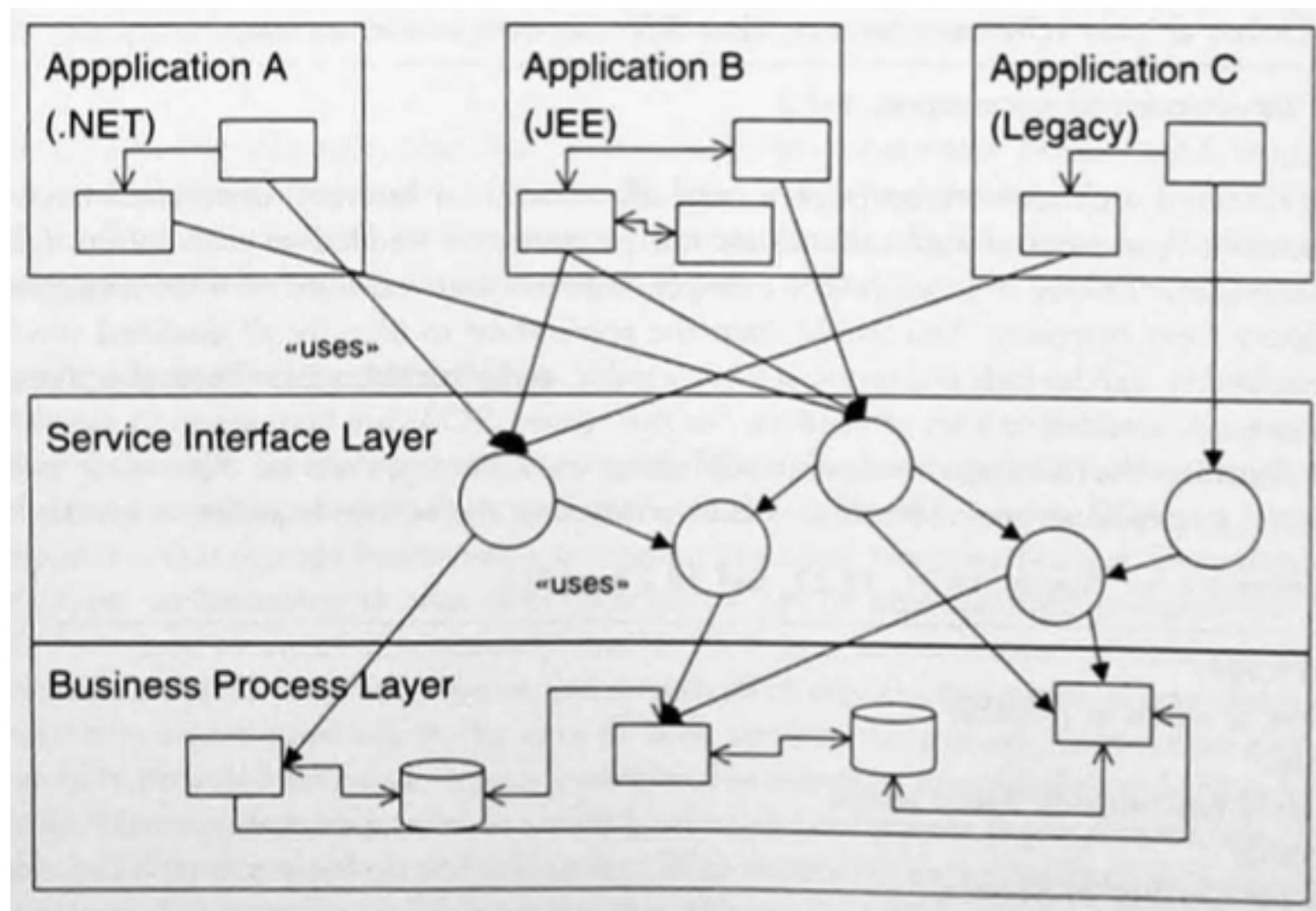
# Microservicios

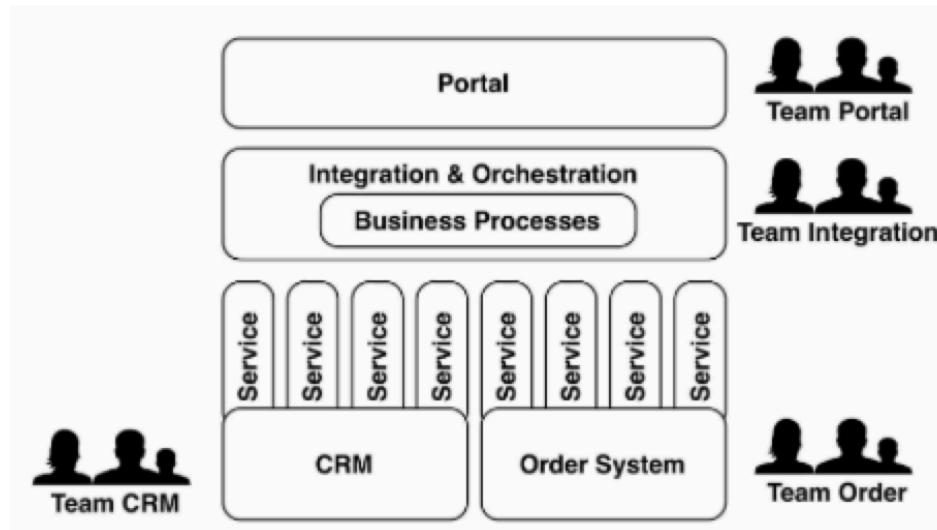
- ▶ Una aplicación se construye en base a un conjunto de pequeños servicios (HTTP, Rest APIs)
- ▶ Servicios pueden escribirse en lenguajes distintos y usar distintos almacenamientos
- ▶ Rompe en pequeños servicios la componente que usualmente corresponde al server
- ▶ En lugar de tener que replicar la aplicación para que escala se replican los servicios que lo requieran

# Características de Arquitectura de Microservicios

- ▶ Componentes son Servicios
- ▶ Altamente cohesivas y desacopladas (smart endpoints, dumb pipes)
- ▶ Usan principios y protocolos de la Web (Http, Rest)
- ▶ Organización en base a capacidades de negocio y no de especializaciones tecnológicas (UI, DBA, middleware)
- ▶ Gobierno descentralizado (un servicio es totalmente independiente)
- ▶ Manejo de datos descentralizado

# SOA





- ▶ CRM - aplicación para manejo de clientes
  - ▶ servicios pueden ser creación de clientes nuevos o información de historia de un cliente
- ▶ Order System - aplicación que maneja órdenes
- ▶ Integration Platform - permite que los servicios interactúen
- ▶ Portal - interfaz para los usuarios

# Compartiendo Código



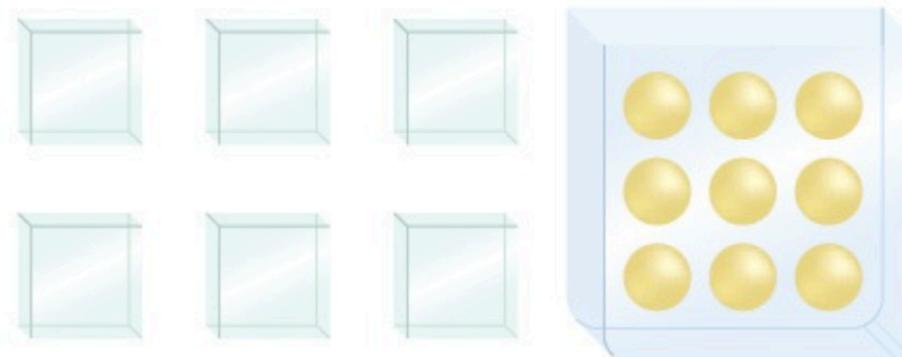
- ▶ En arquitectura clásica un proceso de negocios es implementado como una aplicación
- ▶ Necesariamente en cada aplicación hay código redundante (20%)
- ▶ Con SOA se puede compartir el código que es común



quantity of overall  
automation logic =  $x$



enterprise with an inventory of standalone applications



quantity of overall  
automation logic = 85% of  $x$

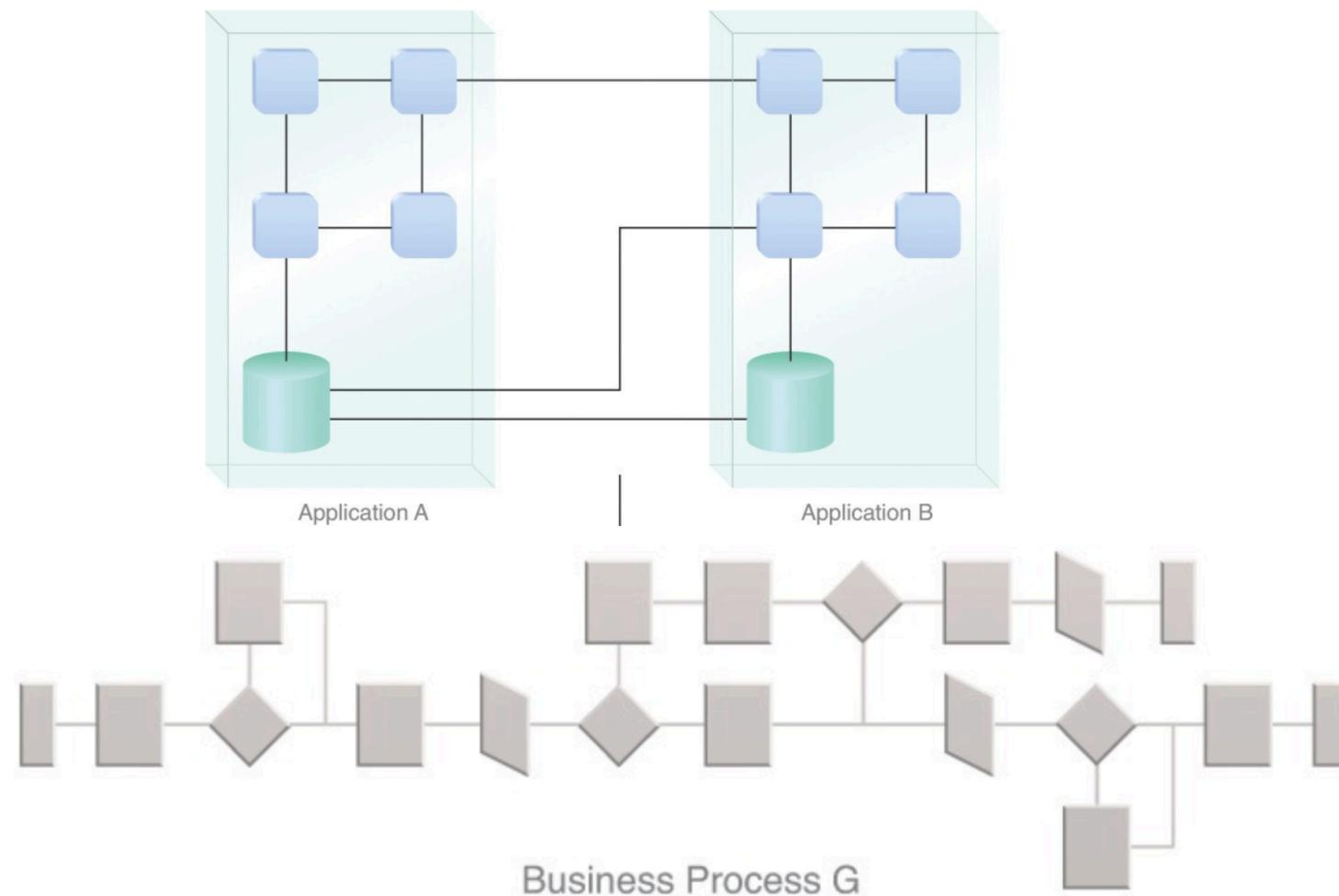
enterprise with a mixed inventory of standalone  
applications and services



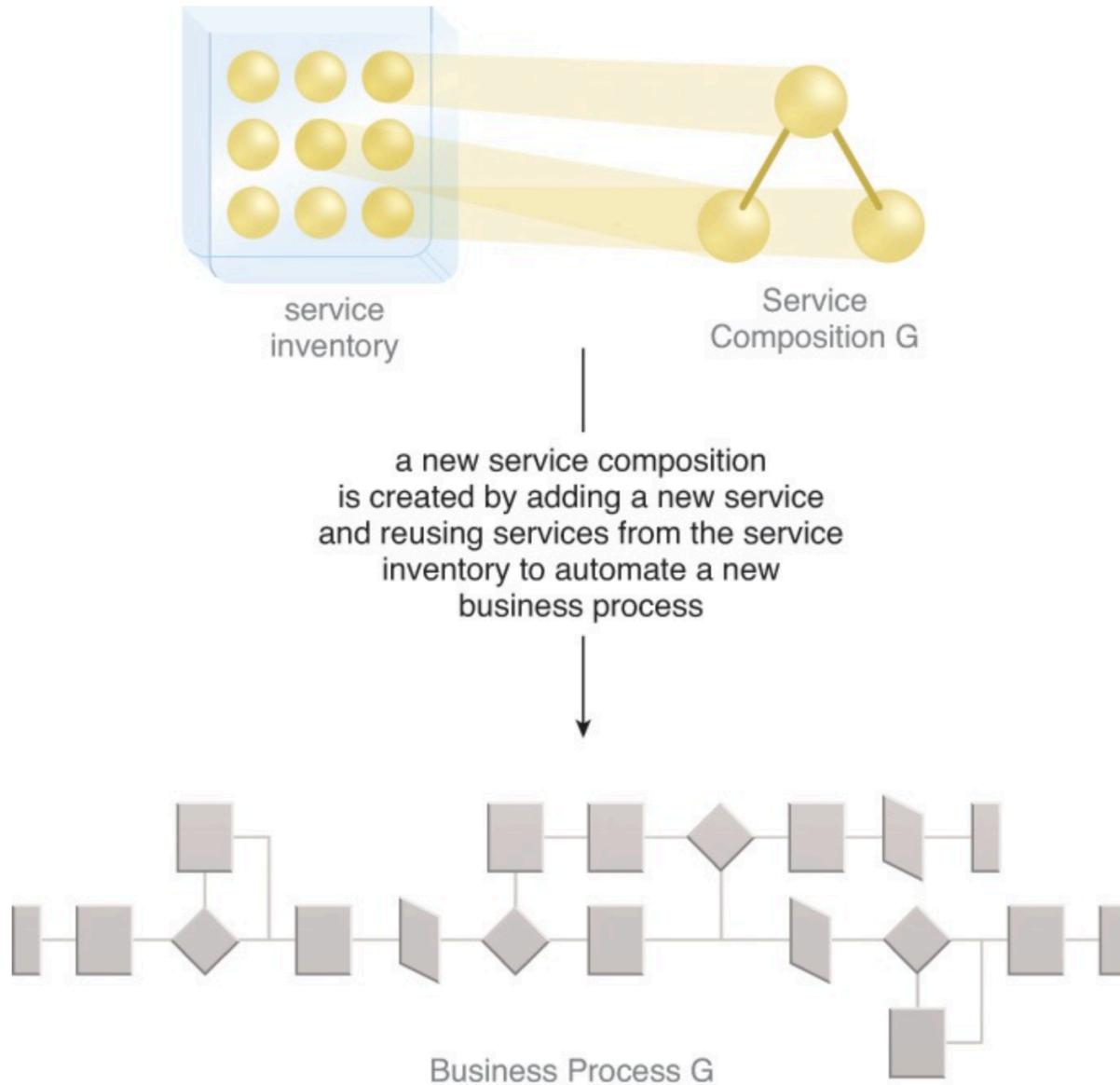
quantity of overall  
automation logic = 65% of  $x$

enterprise with an inventory of services

# A veces un proceso de negocio requiere integrar aplicaciones

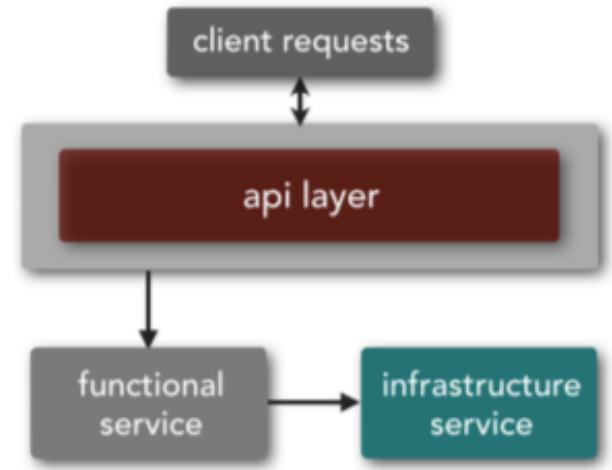


# Con Arquitectura de Servicios



# La llegada de los microservicios

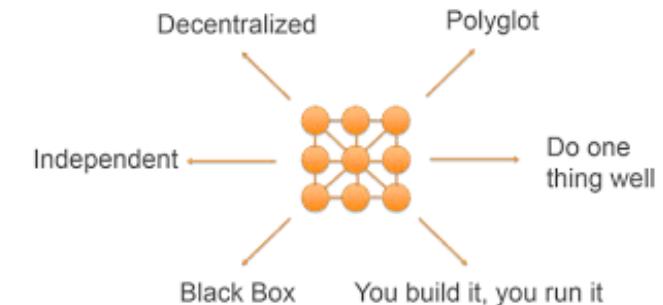
- ▶ 2011 - 2012
- ▶ algunos se refieren a "SOA hecho bien"
- ▶ primera generación de SOA comenzó a hacerse demasiado compleja
  - ▶ SOAP -> middleware
- ▶ representa una maduración de las ideas de SOA a la luz de la experiencia práctica
- ▶ no requiere capa de middleware
  - ▶ cada servicio expone una API (contrato)
- ▶ la mayor parte son servicios funcionales con algunos servicios de infraestructura (no se exponen al mundo externo)



# Características de Arquitectura de Microservicios

- ▶ Componentes son Servicios
- ▶ Altamente cohesivas y desacopladas (smart endpoints, dumb pipes)
- ▶ Usan principios y protocolos de la Web (Http, Rest)
- ▶ Organización en base a capacidades de negocio y no de especializaciones tecnológicas (UI, DBA, middleware)
- ▶ Gobierno descentralizado (un servicio es totalmente independiente)
- ▶ Manejo de datos descentralizado

# Microservicios vs SOA



- ▶ granularidad mucho menor
- ▶ No hay necesidad de capa de integración/orquestación
- ▶ Cada microservicio es responsable de comunicarse con quien lo requiera (mensaje simple sin inteligencia)
- ▶ UI es parte del microservicio
- ▶ Alcance no tiene que ser la organización completa

2000's

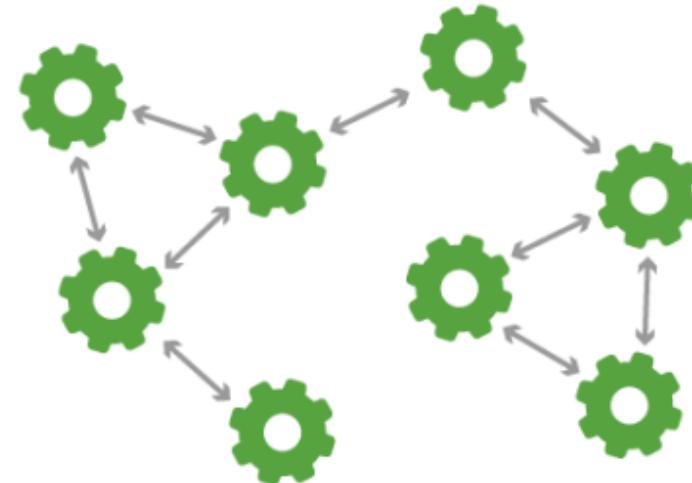
## SERVICE ORIENTED ARCHITECTURE



**SOA** based applications are comprised of more loosely coupled components that use an Enterprise Services Bus messaging protocol to communicate between themselves.

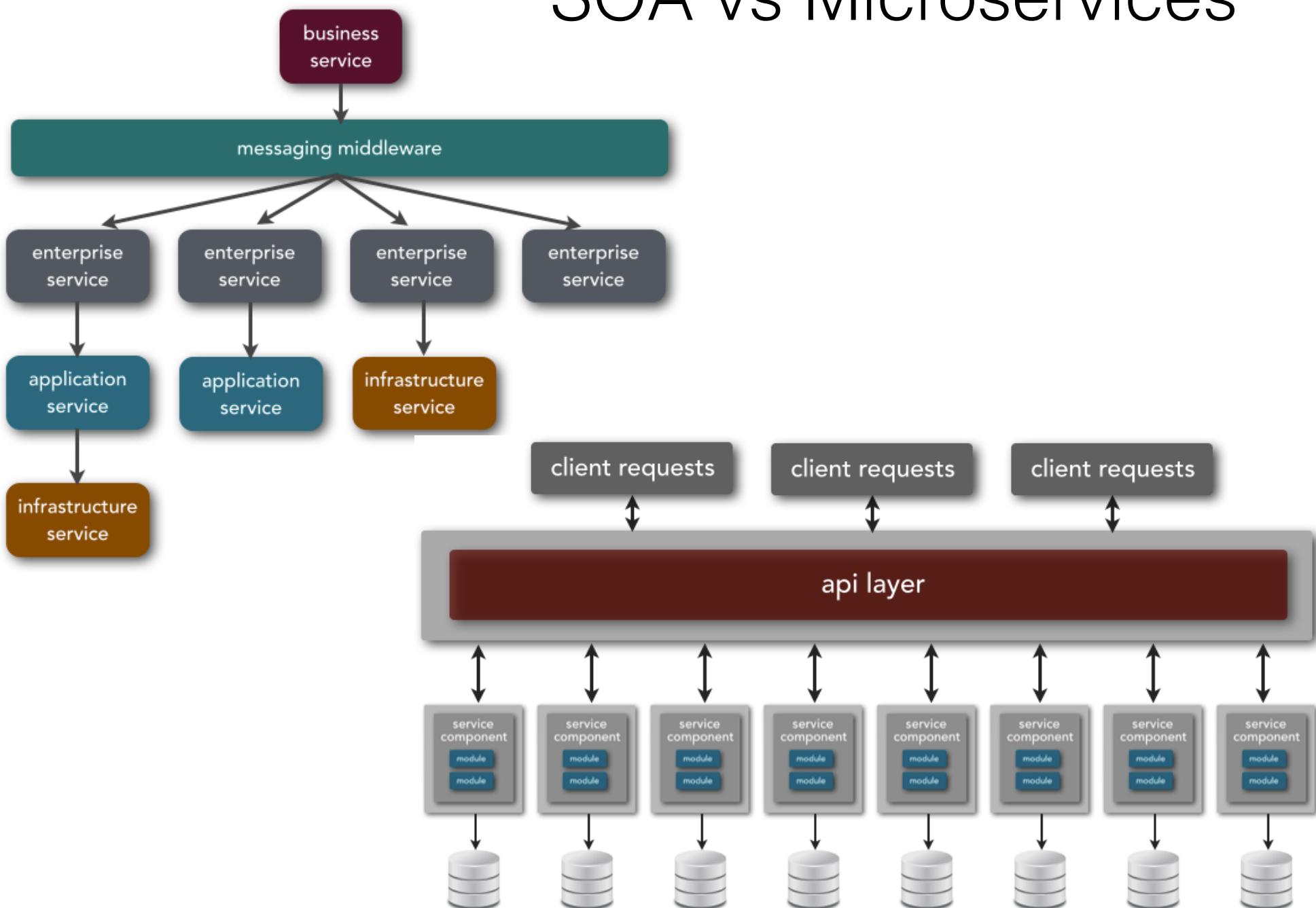
2010's

## MICROSERVICES ARCHITECTURE

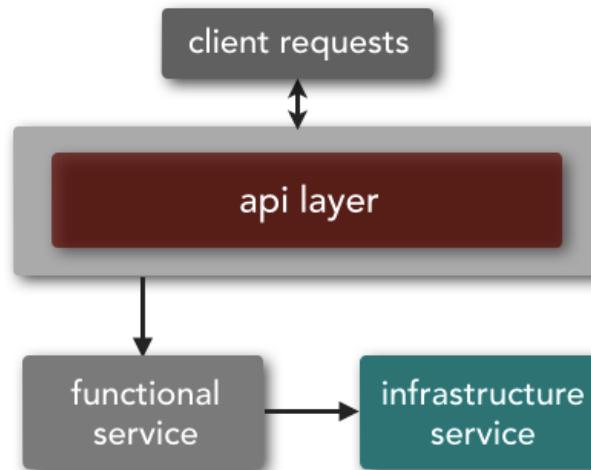


**Microservices** are a number of independent application services delivering one single functionality in a loosely connected and self-contained fashion, communicating through light-weight messaging protocols such as HTTP, REST or Thrift API.

# SOA vs Microservices



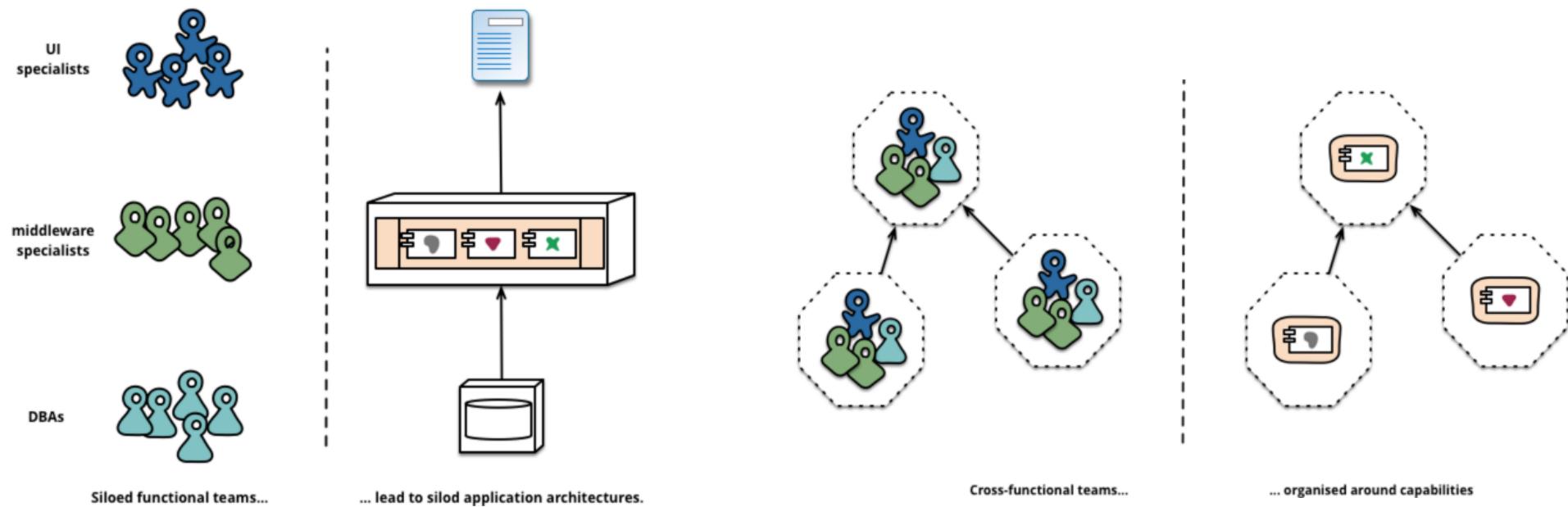
# Tipo de Microservicios



- ▶ Servicios funcionales - Se exponen hacia afuera
- ▶ Servicios de infraestructura - Para uso interno (de los otros servicios)
  - ▶ autenticación y autorización
  - ▶ monitoreo
  - ▶ logging y auditoría

# Organización de Equipo de Trabajo

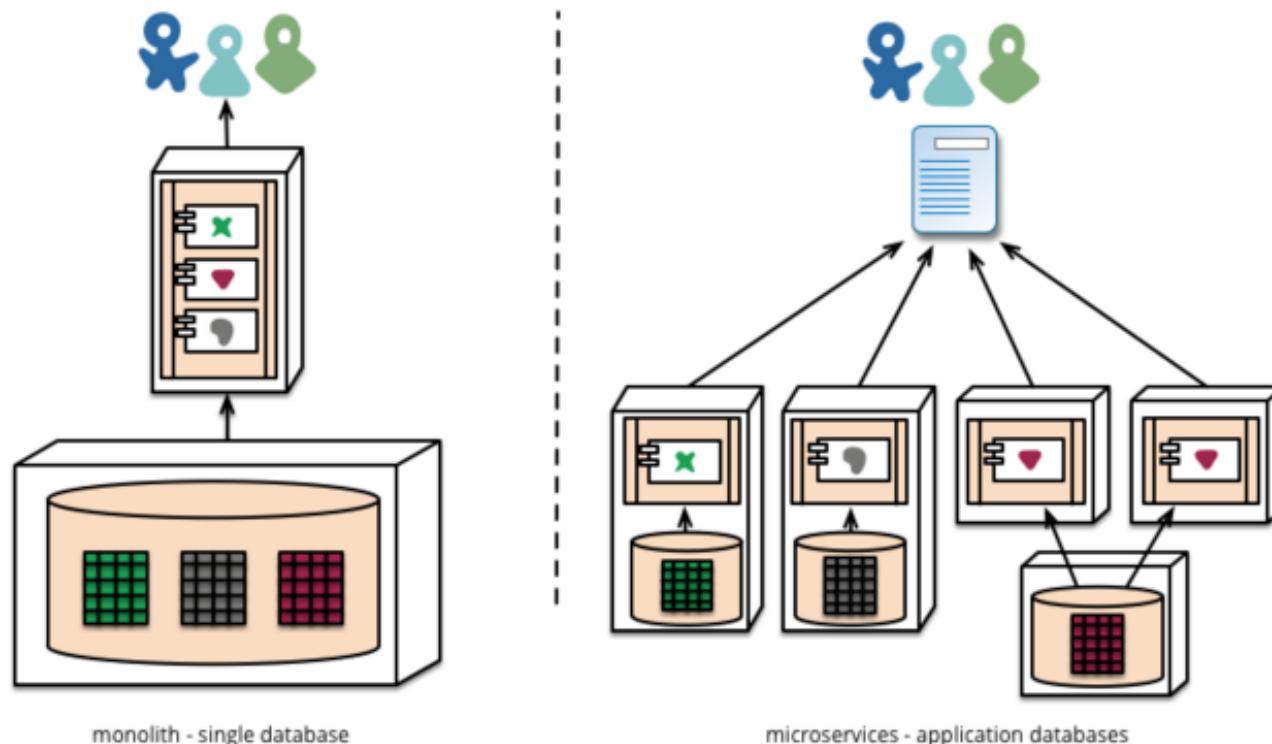
- enfoque clásico es generar equipos de front-end, back-end, bases de datos
- cambios requieren que equipos conversen



**Clásico: front-end, back-end, DB**

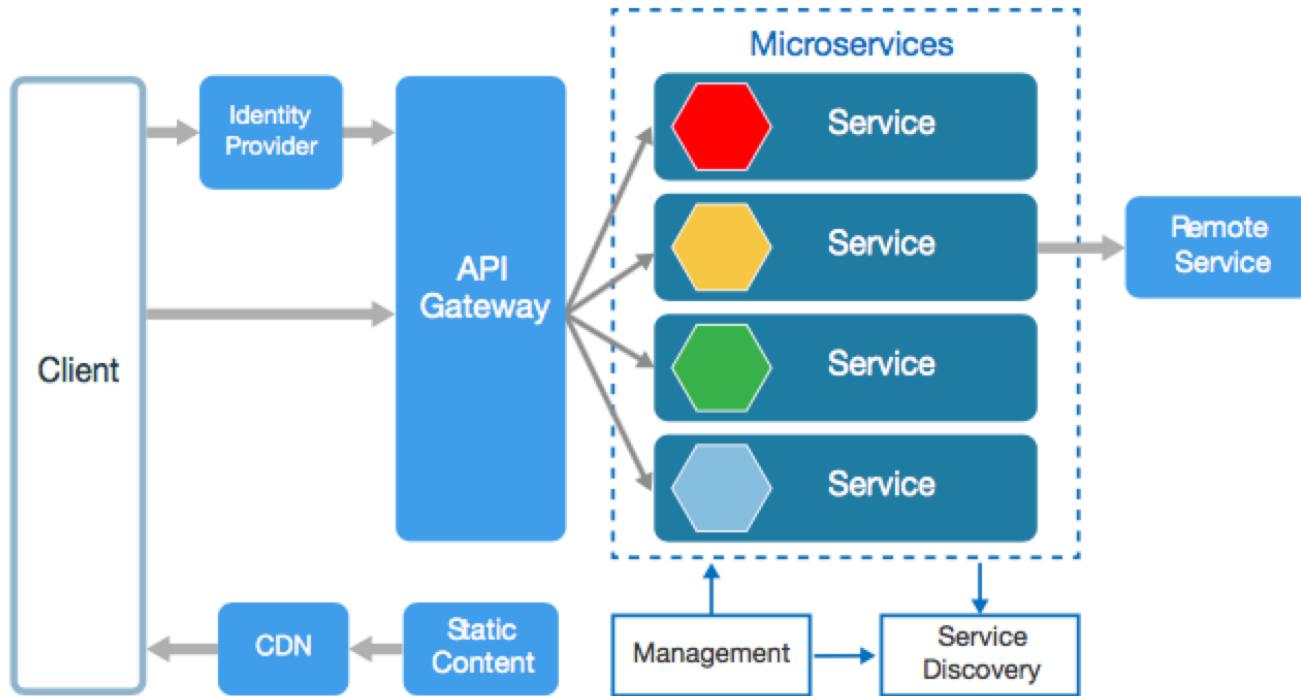
**Microservices: equipos multifuncionales**

# Manejo de Datos Descentralizado



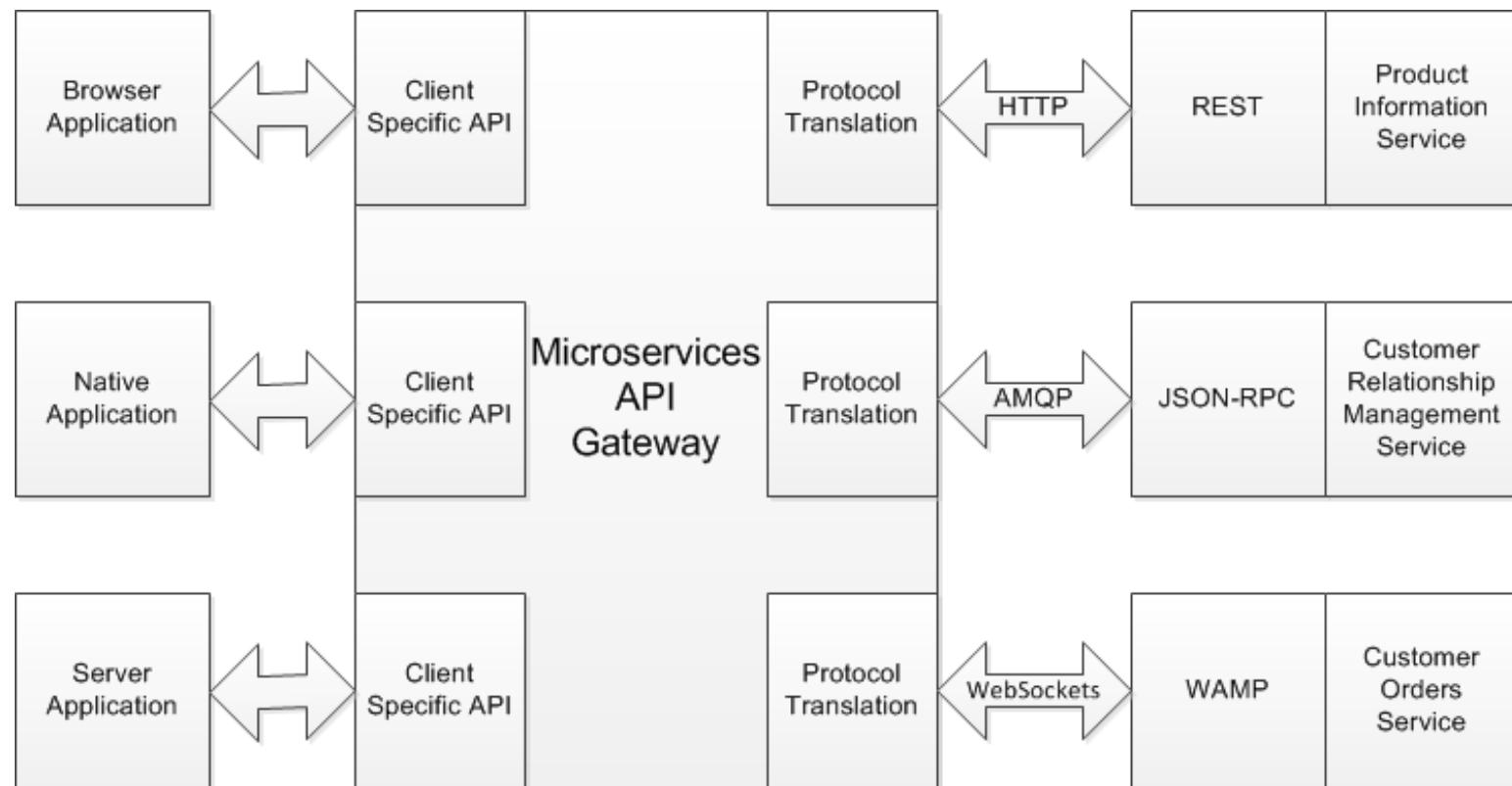
- cada servicio maneja su propia BD
- pueden ser instancias de una BD o tecnologías completamente diferentes
- no se usan transacciones para coordinar, consistencia eventual
- respuesta rápida y escalabilidad se paga con posibles grados de inconsistencia temporal

# El API Layer



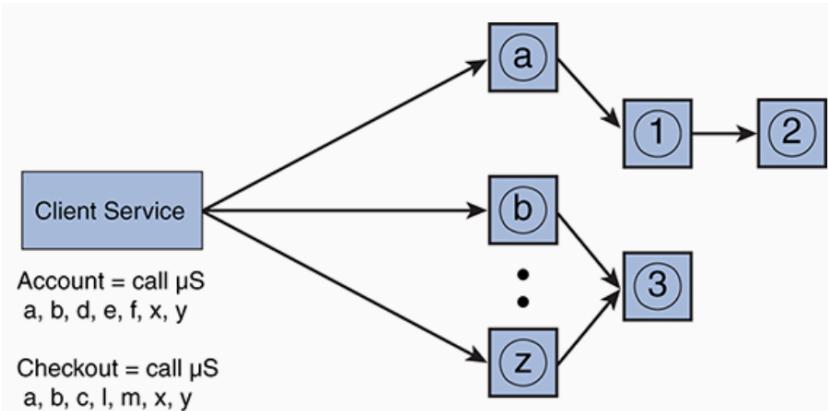
- lo que se expone a clientes
- puede agregar respuestas de varios servicios
- permite cambiar interfaz de servicios
- servicios pueden usar protocolos que no son http
- puede proporcionar servicios generales (logging, load balancing, etc)

# API Gateway

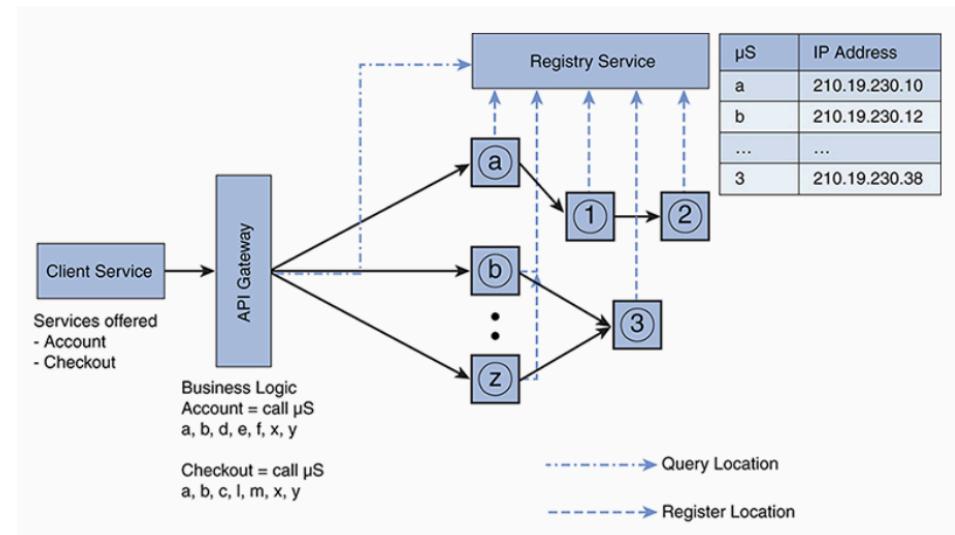


# Variaciones

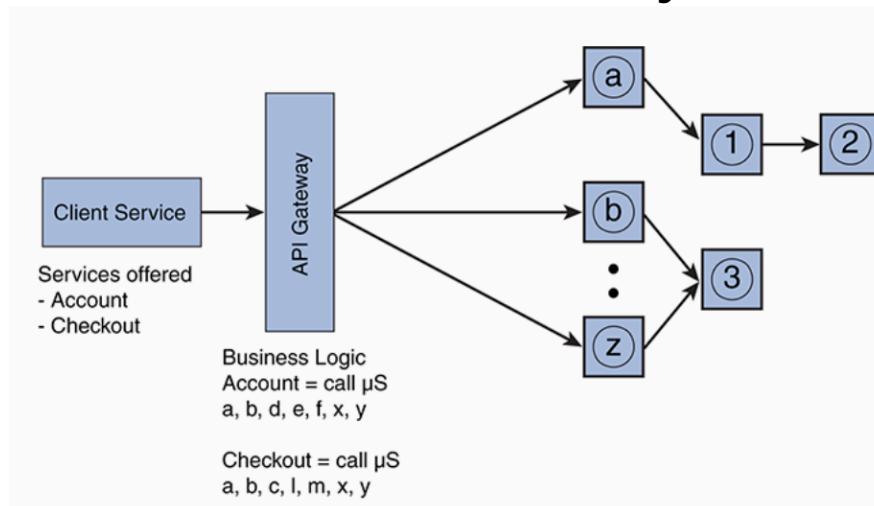
## Sin API Gateway



## Con API Gateway y Registry



## Con API Gateway



# No es la panacea

panacea

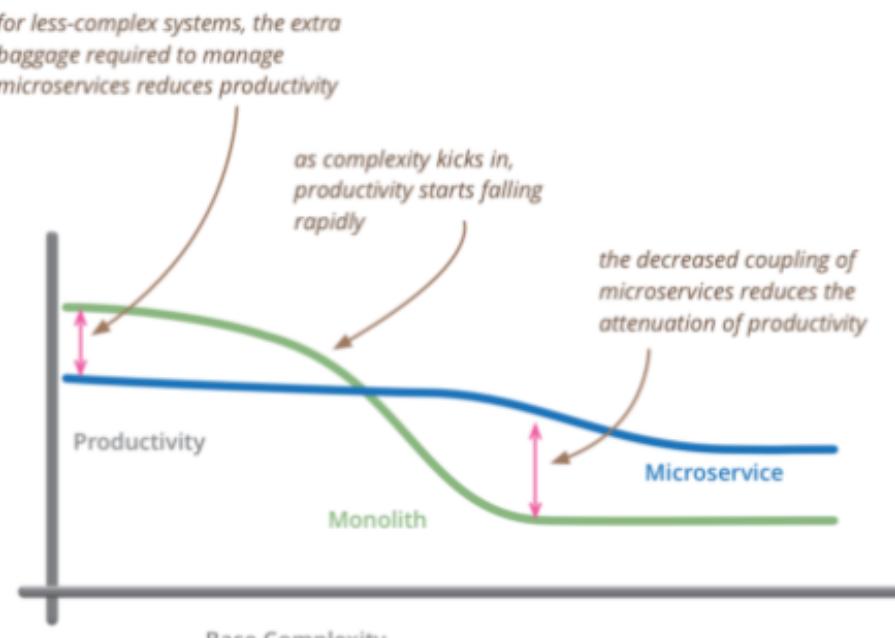
/,panə'si:ə/ ⓘ

*noun*

a solution or remedy for all difficulties or diseases.

"the panacea for all corporate ills"

*synonyms:* universal cure, **cure-all**, cure for all ills, universal remedy, sovereign remedy, **heal-all**, **nostrum**, **elixir**, wonder drug, perfect solution, magic formula, **magic bullet**; **More**



*but remember the skill of the team will outweigh any monolith/microservice choice*

# Desafíos

- ▶ complejidad puede ser mayor que el de la equivalente app monolítica
- ▶ problemas de gobernanza
- ▶ congestión de la red y latencia
- ▶ integridad de los datos
- ▶ requiere pensar en forma distribuida

# Mejores prácticas

- ▶ modelar servicios de acuerdo a dominio de negocio
- ▶ alta coherencia y bajo acoplamiento
- ▶ descentralizar todo (equipos, esquemas, código)
- ▶ datos privados para cada servicio
- ▶ comunicación a través de apis bien definidas
- ▶ api gateway no debe saber nada del dominio