CHAPTER 10

Arquitecturas Comunes y Patrones Arquitectónicos

La idea de un patrón arquitectónico es similar a la de un patrón de diseño. Se trata de una solución a un problema en contexto. Una solución que suele usarse muchas veces en la práctica y que ha demostrado tener beneficios. Al igual que en el caso de los patrones de diseño, permiten no tener que partir de cero.

10.1 Arquitectura Cliente-Servidor o Multi-Tier

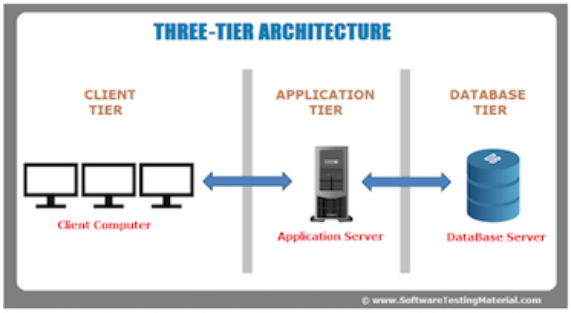
En esta arquitectura la responsabilidad se divide entre dos grandes *containers*: el cliente y el servidor (en *client-server*) o en más de dos (*Tiered*). Cada uno de estos *Tiers* son usualmente deployados en máquinas distintas, pero no necesariamente tiene que ser así.

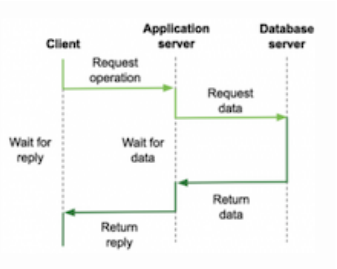
La palabra "*tier*" tiene el significado de nivel, pero ello es algo engañoso en este caso porque nivel conlleva una idea de ordenamiento (nivel más alto que otro) lo que no es así en este caso.

La arquitectura *client-server* fue una revolución en los 80's y 90's puesto que antiguamente la totalidad de la responsabilidad la asumía una sola máquina (el *mainframe*). La ventaja de separar en cliente y servidor es que es posible escalar más fácilmente el número de usuarios y además permitió el uso de interfaces gráficas que en el modelo del mainframe habría significado sobrecargar la máquina.

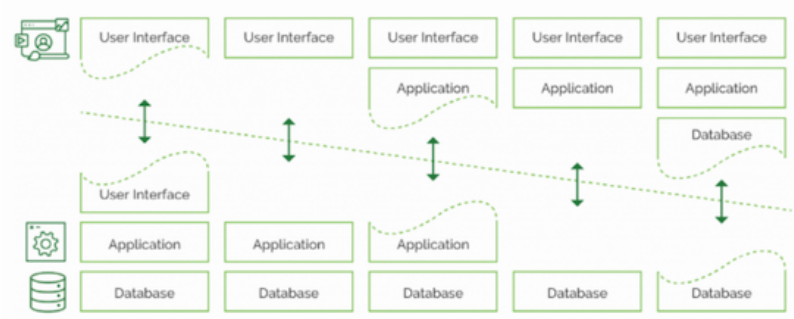
La arquitectura cliente-servidor básica evolucionó a una arquitectura de 3 *tiers* en que el cliente es responsable solamente de la interfaz de usuario. Esto

se suele llamar cliente delgado o "*thin client*". La figura muestra la forma en que funciona una arquitectura de 3 niveles.



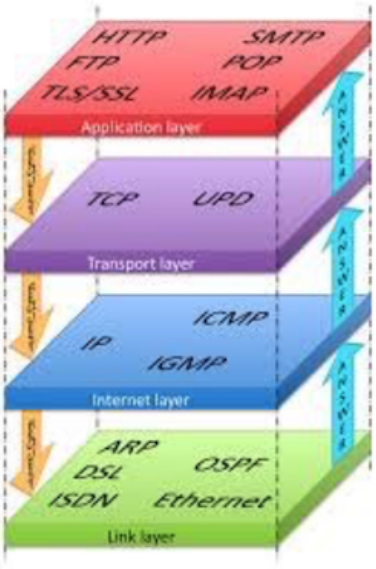


La figura muestra posibles variaciones en las responsabilidades que toma el cliente, desde solo una parte de la interfaz de usuario, toda la interfaz de usuario, toda la interfaz y parte de la lógica de la aplicación, toda la interfaz y toda la lógica o incluso todo lo anterior y parte del acceso a la base de datos.



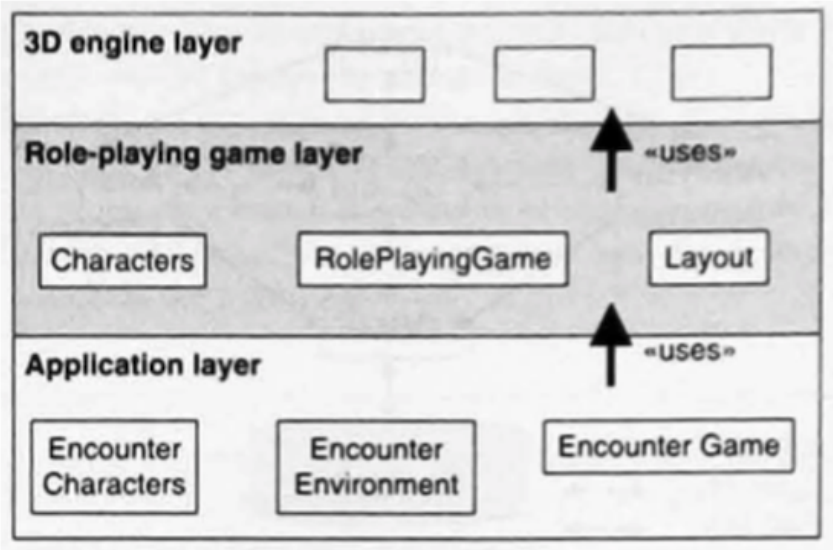
10.2 *Layered Architecture* (Arquitectura de capas)

Esta arquitectura se suele confundir con la anterior. En este caso la idea de capas es en realidad que existe un ordenamiento (hay capas inferiores y superiores) y normalmente todas las capas son parte de una unidad que corre en una misma máquina.



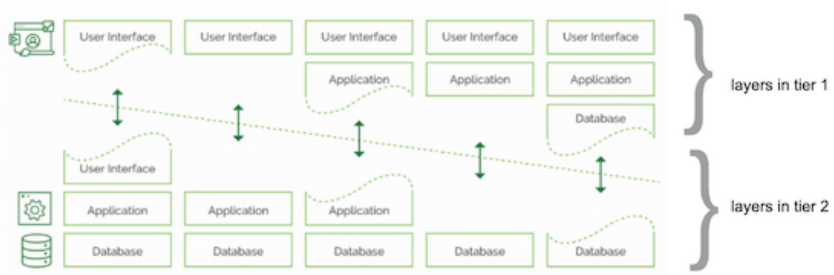
El gran problema que resuelve la arquitectura de capas es poder manejar la complejidad de una pieza de software. La idea es que al pensar en como implementar algo en lugar de pensar en elementos de bajo nivel puedo utilizar abstracciones de más alto nivel como si ya existiesen. Por ejemplo, en el diseño de la Web se pensó en un protocolo (HTTP) pero este protocolo asume que existe TCP y TCP asume que existe IP. Finalmente, IP usa la capa específica de la red considerada.

Un segundo ejemplo puede ser la arquitectura de un juego de computador que tiene una capa de aplicación que usa la capa de *Role-Playing* que a su vez asume que existe una capa que implementa el motor de *rendering* 3D.



Es posible que estos patrones arquitectónicos aparezcan combinados. Por

ejemplo, podemos tener una arquitectura de capas en dos *tiers*

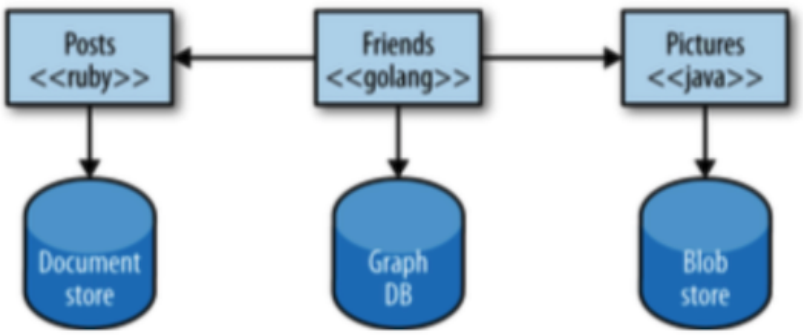


10.3 Arquitectura de Servicios

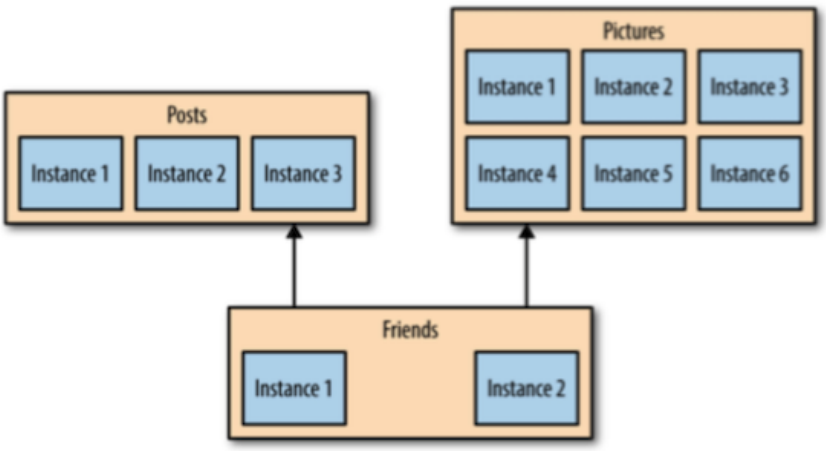
Hemos visto que la arquitectura tiene que ver con separación del sistema en sus grandes constituyentes. En un primer nivel contenedores y en un segundo nivel componentes. También dijimos que una componente es una unidad que puede ser reemplazada o mejorada, cuidando solo de mantener la interfaz.

En las arquitecturas clásicas una componente se implementa como un módulo, un paquete o una librería compartida que debe ser ensamblada con el todo en el momento del *build* o del *deploy*. En los últimos años, se ha estado haciendo muy popular el que las componentes correspondan a servicios que corren en un proceso totalmente distinto (incluso puede ser en otra máquina) y que se comunica con el resto del sistema usando protocolos como http. Se dice que en este caso las componentes están débilmente acopladas.

El ejemplo siguiente muestra una aplicación que incluye tres grandes componentes que se han implementado como servicios: *Posts* que está implementada en Ruby y utiliza un motor de BD con modelo de documentos para administrar los posteos de los usuarios, *Friends* que está implementada en Go y es responsable de manejar la red social para lo que se vale de un motor de grafos, y finalmente *Pictures* que se encarga de gestionar las imágenes que son posteadas usando un almacenamiento especializado para este tipo de objetos.



Una de las ventajas de la arquitectura de servicios es que cada servicio puede ser desarrollado usando la plataforma que uno quiera y puede usar su propia base de datos. Otra gran ventaja es que cada uno de los servicios puede escalar en forma independiente. Por ejemplo, el servicio de *Posts* requiere de 3 instancias (máquinas virtuales) en cambiom el servicio de *Pictures* requiere 6 instancias.



Una arquitectura orientada a servicios tiene muchas ventajas interesantes:

- Cada componente puede ser deployada en forma completamente independiente y si se decide actualizarla solo requiere redeployar esa componente.
- La existencia de una interfaz explícita muy clara hace más difícil que con el tiempo se violen acuerdos
- La posibilidad de usar distintas tecnologías. Cada servicio puede ser desarrollado en la plataforma que presente ventajas o para la cual exista experiencia del equipo.
- Cada componente puede ser escalada en forma independiente (ver figura anterior)
- El sistema como un todo puede hacerse más resiliente, ya que una falla afecta solo a un servicio. Si se quiere aún mayor resiliencia, pueden duplicarse los servicios críticos.

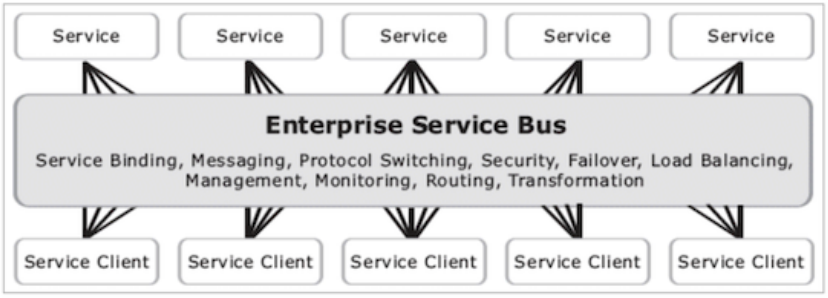
Por supuesto, como siempre, existen compromisos (*tradeoffs*) y existen también problemas con este enfoque.

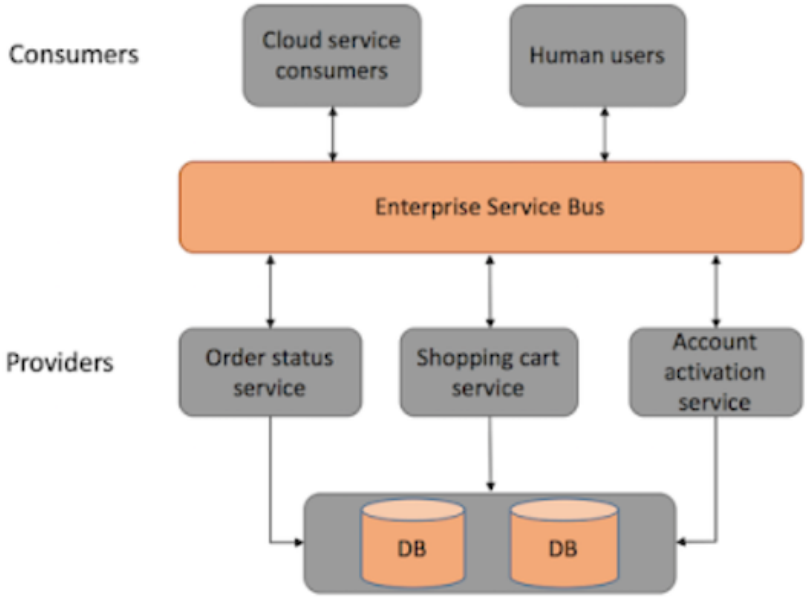
- Puede haber un compromiso de desempeño por el aumento en el intercambio de mensajes (http u otros) que son muchísimo más lentos que una invocación de un procedimiento interno. El poner más instancias no va a aminorar este problema.
- Puede resultar en un mayor tiempo o esfuerzo de desarrollo (sobre todo si el equipo tiene poca experiencia en arquitecturas de servicios)

La arquitectura orientada a servicios clásica (SOA)

A mediados de los 2000 y muy influidos por la disponibilidad de los Web Services (SOAP, WSDL) comenzaron a aparecer las primeras arquitecturas que implementaban esta idea de que las componentes iban a ser servicios y que el sistema no era otra cosa que una combinación de servicios. Al mismo tiempo, la formalización de los procesos de negocio en forma de diagramas BPMN hizo que muchas organizaciones trataran de asimilar los procesos de negocio con servicios al construir sistemas de información.

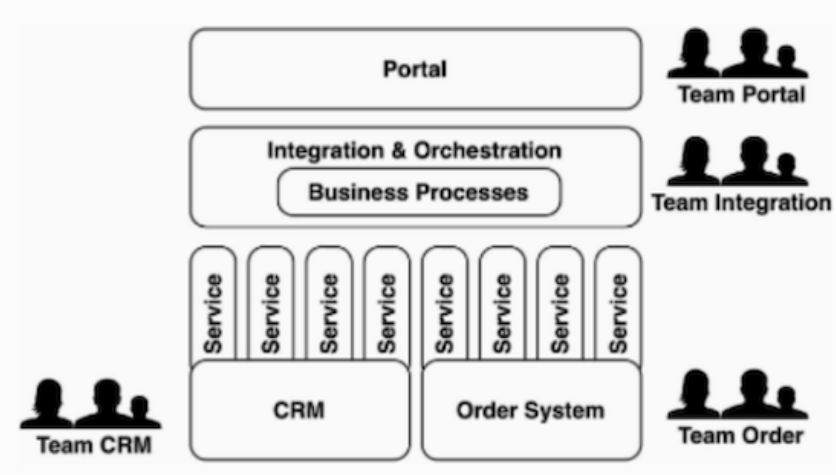
Esta primera generación de arquitecturas de servicio concibe los servicios íntimamente asociados a los procesos de negocio y tienden a ser de una granularidad más bien gruesa (servicio de órdenes de compra, servicio de carro de compra, etc). También es común en esta arquitectura que los servicios interactúen a través de una unidad denominada *enterprise service* bus que se encarga de combinarlos adecuadamente y proveer aspectos necesarios para poder llevar a cabo transacciones complejas. Los clientes no acceden directamente a los servicios, sino que lo hacen a través del bus de servicios.





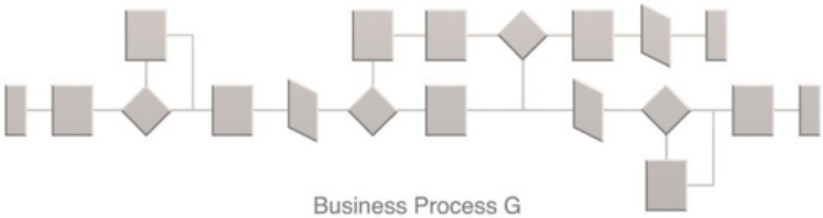
De esta forma, un CRM o un sistema para el manejo de órdenes de compra

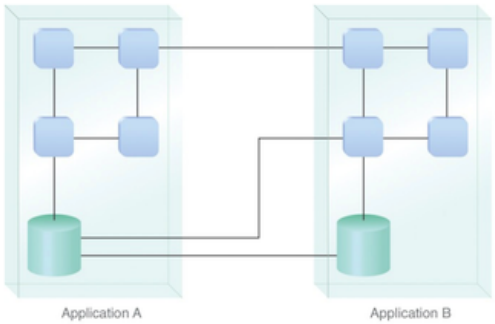
puede ser visto como una pieza de software que incorpora un cierto número de servicios para lograr su objetivo. Dado que los servicios del CRM y del sistema de órdenes comparten el bus de servicios, es posible integrar con facilidad funcionalidades de los sistemas y permitir por ejemplo que usuarios accedan a una funcionalidad integrada a través de un portal.



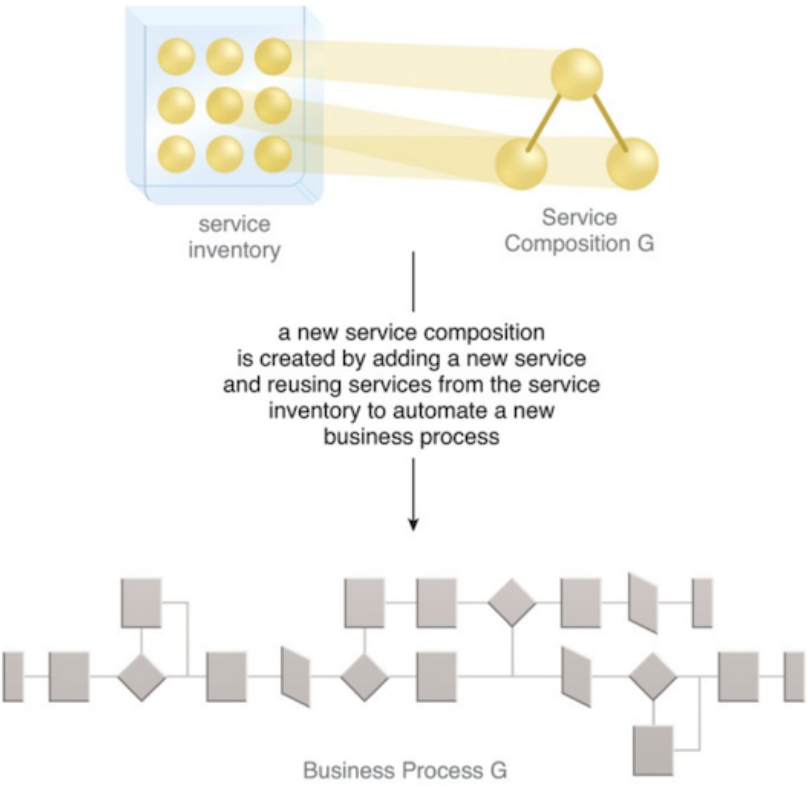
Otra gran ventaja que entrega a una organización usar una arquitectura orientada a servicios es que se reduce la cantidad de código necesario para implementar las funcionalidades requeridas por sus procesos de negocio. Esto es así porque en un enfoque de aplicaciones independientes siempre hay partes del código que se repite o es común (por lo menos un 20%). Al separar en servicios, ese código común puede ser reutilizado entre ellos. En casos reales se ha visto una disminución de hasta un 35% del código si la organización lleva todas sus aplicaciones a una arquitectura de servicios.

El tener que integrar aplicaciones es un problema a enfrentar y donde una arquitectura de servicios facilita enormemente las cosas. La necesidad de integrar aplicaciones surge del hecho que hay procesos de negocio que trascienden una sola aplicación, como ilustra la siguiente figura:





Si en lugar de tener aplicaciones lo que tenemos es un inventario de servicios, un proceso de negocios en particular va a requerir solo una composición particular de dichos servicios

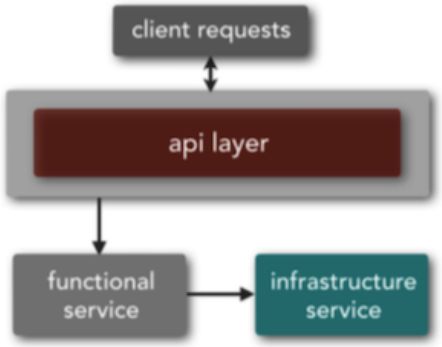


La llegada de los microservicios

A pesar de las enormes ventajas asociadas a SOA, y luego de numerosos proyectos en que hubo muchos fracasos, se comenzó a criticar principalmente por la complejidad asociada a la implementación del bus de servicios, la necesidad de incorporar piezas de software adicionales (*middleware*) y la proliferación de nuevos estándares mucho más allá de los asociados a los Web Services (SOAP, WSDL, UDDI) que tenían que ver con seguridad, orquestación, etc.

A comienzos de la década del 2010 aparece entonces una reformulación de la idea en una arquitectura que se denomina de microservicios. Esta arquitectura representa una maduración de las ideas de la primera generación y que responde a las principales críticas que se estaban formulando. Las principales características diferenciadoras respecto a SOA son las siguientes:

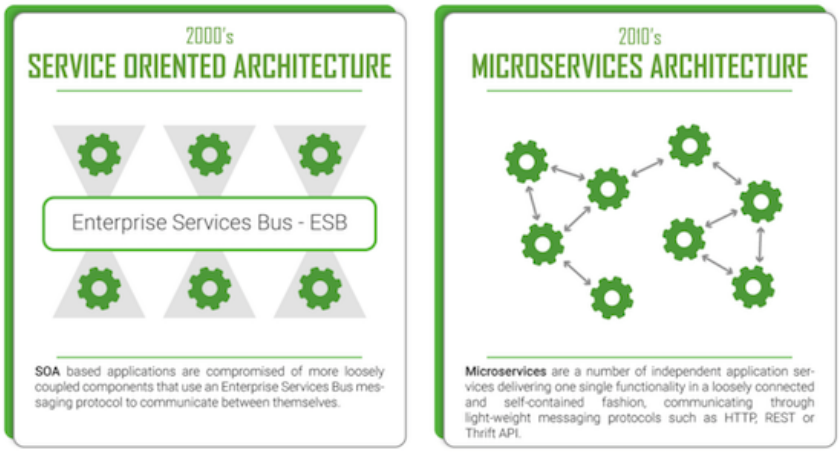
- La granularidad de los servicios es mucho menor. Esto facilita la reutilización de estas componentes, ya que al ser demasiado grandes muchas veces no servían para incorporarlos en más de una aplicación.
- Desaparece la necesidad del bus de servicios y, por lo tanto, de un *middleware* que permita conectar o comunicar los servicios
- Tienden a usar el protocolo REST en lugar de SOAP para interactuar



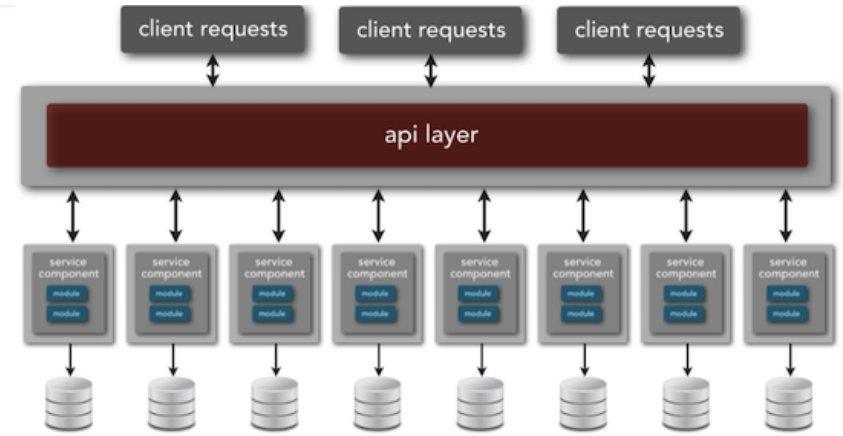
Los servicios exponen una API que puede ser usada directamente por los clientes, o bien varias APIs pueden ser combinadas para exponer una API conjunta (similar a la idea de una fachada) hacia el cliente. Asimismo, es posible que algunos servicios no colaboren a esta capa de APIs y solo sean utilizados por otros servicios. A ellos se les denomina servicios de infraestructura en contraposición a los servicios funcionales. Ejemplos típicos de servicios de infraestructura son servicios de autenticación, monitoreo y auditoría.

Bajo la arquitectura de microservicios, una aplicación se construye en base a un conjunto de pequeños servicios (HTTP, Rest APIs) que pueden escribirse en lenguajes distintos por equipos de desarrollo diferentes y usar distintos almacenamientos. La componente que usualmente corresponde al *server* es separada en pequeños servicios, los que pueden ser replicados en forma independiente si es necesario.

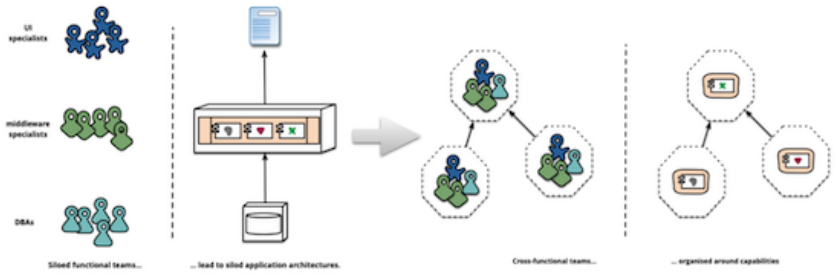
Si comparamos la arquitectura SOA de los 2000's con la de microservicios de los 2010's vemos que la idea de que las aplicaciones sean construidas en base a componentes débilmente acopladas (servicios) y que usan un bus de servicios para implementar el protocolo que les permite colaborar, es llevada a un número de servicios más pequeños y totalmente independientes y autocontenidos que conversan directamente usando protocolos como HTTP y REST.



Aunque no es estrictamente necesario, es común que se establezca una *API Layer* que comprende realmente las funcionalidades que serán expuestas a los clientes

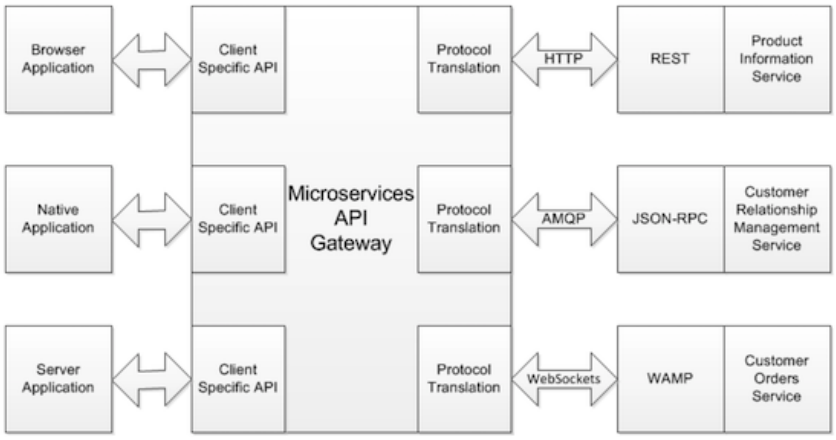


Una consecuencia del uso de una arquitectura de servicios es que los equipos de desarrollo deben ser multifuncionales, puesto que normalmente les tocará desarrollar un servicio en su totalidad. Esto es una ventaja con los procesos de desarrollo modernos como SCRUM en que se recomienda constituir equipos multifuncionales.



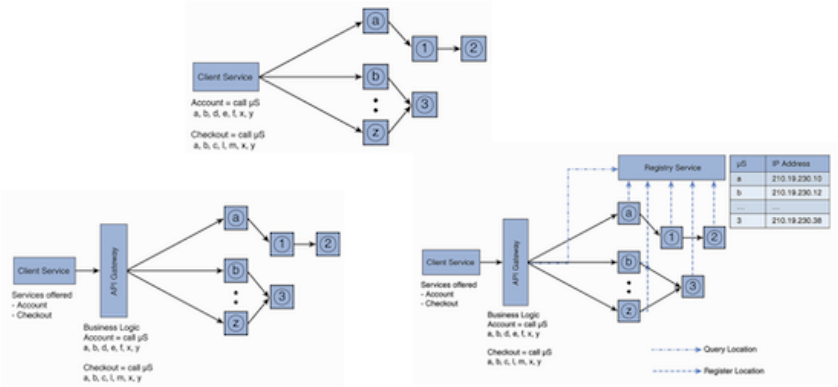
El manejo de datos pasa a ser descentralizado, puesto que cada servicio es responsable de su propia base de datos. Esto permite que se usen tecnologías completamente diferentes (MySQL en una y MongoDB en otra). Dado que no existe un ente centralizador como el bus de servicios, no hay soporte de transacciones complejas y por lo general se aceptan modelos de consistencia eventual.

Volvamos ahora al tema del *API Layer*, tambien llamado *API Gateway*, que actúa como una fachada de los servicios hacia el lado del cliente y que representa lo que realmente queda expuestos de los servicios. Este *API Gateway* no solo simplifica para el cliente el uso de los servicios, sino que permite cambiar los protocolos de comunicación. Por ejemplo, un servicio puede tener una API que requiere un protocolo que no es HTTP y que puede ser transformado por el *API Gateway* de modo que los clientes usen HTTP

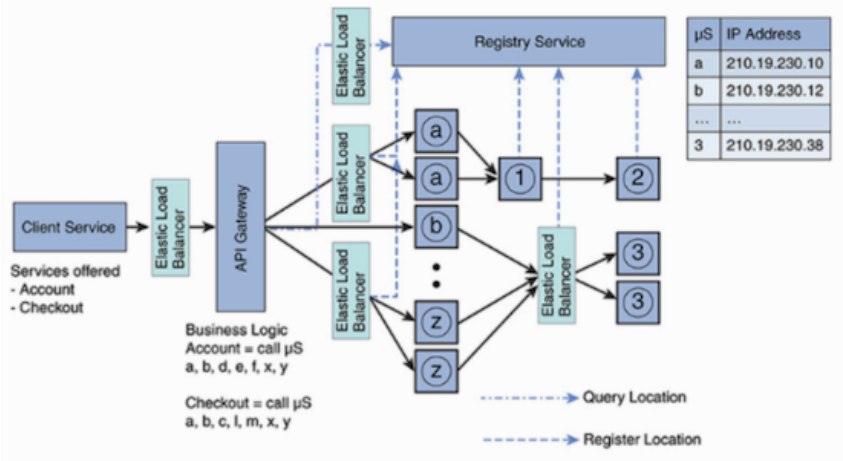


Cuando el inventario de servicios es muy grande, se puede incorporar un servicio de registro (*Registry*). En la figura puede verse a través de un ejemplo sencillo la diferencia entre tener o no tener un *Gateway* y un *Registry*.

Los servicios que el cliente requiere son dos: *account* y *checkout*. En el primer caso (sin *API Gateway*) el cliente debe conocer muy bien los servicios que debe utilizar, puesto que accederá a ellos en forma directa. En el segundo caso (con *API Gateway*) el *Gateway* muestra al cliente una interfaz exactamente como lo requiere, con dos operaciones. Finalmente, la última figura solo agrega el servicio que permite saber exactamente la dirección de cada uno de los servicios disponibles.



Finalmente, si necesitamos mantener múltiples instancias de algunos servicios, es posible que se requiera agregar balanceadores de carga que distribuyan la pega



A pesar de las enormes ventajas de la arquitectura de servicios, hay que decir nuevamente que no es la panacea. Primeramente en términos de productividad del equipo de desarrollo solo hay ganancias cuando se trata de una aplicación relativamente compleja, ya que en escenarios simples es mucho más rápido construir una versión clásica o monolítica. En segundo término, dado que los servicios son pequeños, la penalidad en desempeño por costo del enorme número de mensajes (HTTP) para lograr una funcionalidad puede ser incluso más severo que en la arquitectura de servicios clásica.

La tabla siguiente resume algunos de los pros y contras de la arquitectura de microservicios ...

Si consideramos solo costos para comparar las arquitecturas monolíticas (M) vs la de microservicios (S), deberíamos considerar el costo de construir, mantener y escalar. El primer costo suele ser más bajo en la versión monolítica, pero los microservicios ganan en las otras dos.

Si lo que nos interesa es reducir el *time-to-market* (TTM), la arquitectura de microservicios es la ganadora puesto que saca mejor partido de lo que ya ex-

iste para agregar nuevas features.

Una organización que decide abrazar una arquitectura de microservicios para sus sistemas de información enfrenta una gran cantidad de desafíos adicionales:

- necesidad de mantener y monitorear cientos de microservicios
- problema de descubrimiento, se debe manejar un inventario y proveer formas de consultar
- como asegurar SLAs cuando hay servicios que dependen de otros
- no es suficiente asegurar correctitud de los servicios
- muchas opciones de escalamiento (flexible pero más complejo)
- decisiones de *upgrading* más complejas
- asegurar seguridad es más complejo

En cualquier caso, una vez que la decisión esté tomada, se recomienda una migración gradual hacia los microservicios y no usar un esquema de "*big bang*" (ver figura)