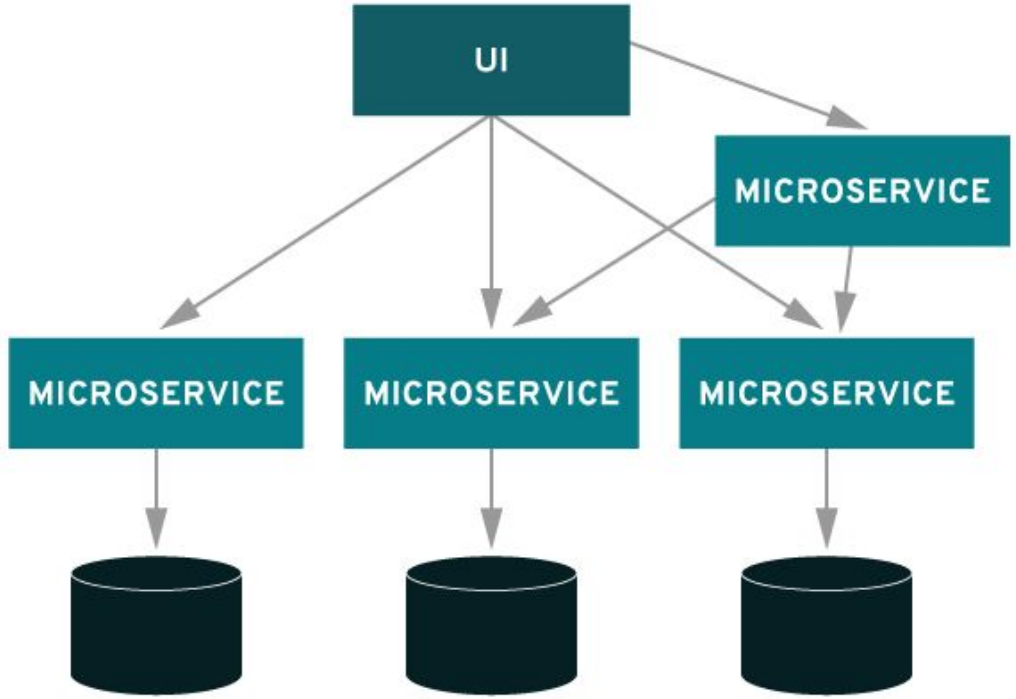
Arquitecturas de Sistemas y Taller de Microservicios



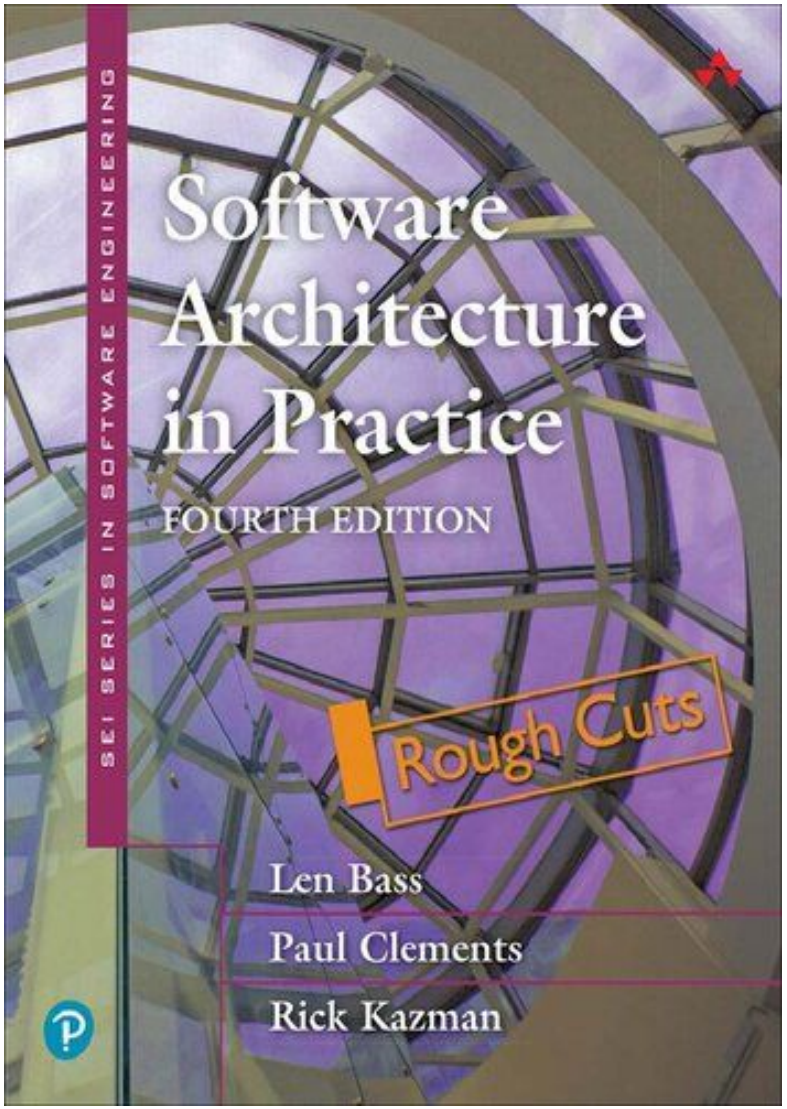
Nicolás Martínez Varsi – Profesor Adjunto
Escuela de Tecnología – Facultad de Ingeniería
Universidad ORT Uruguay

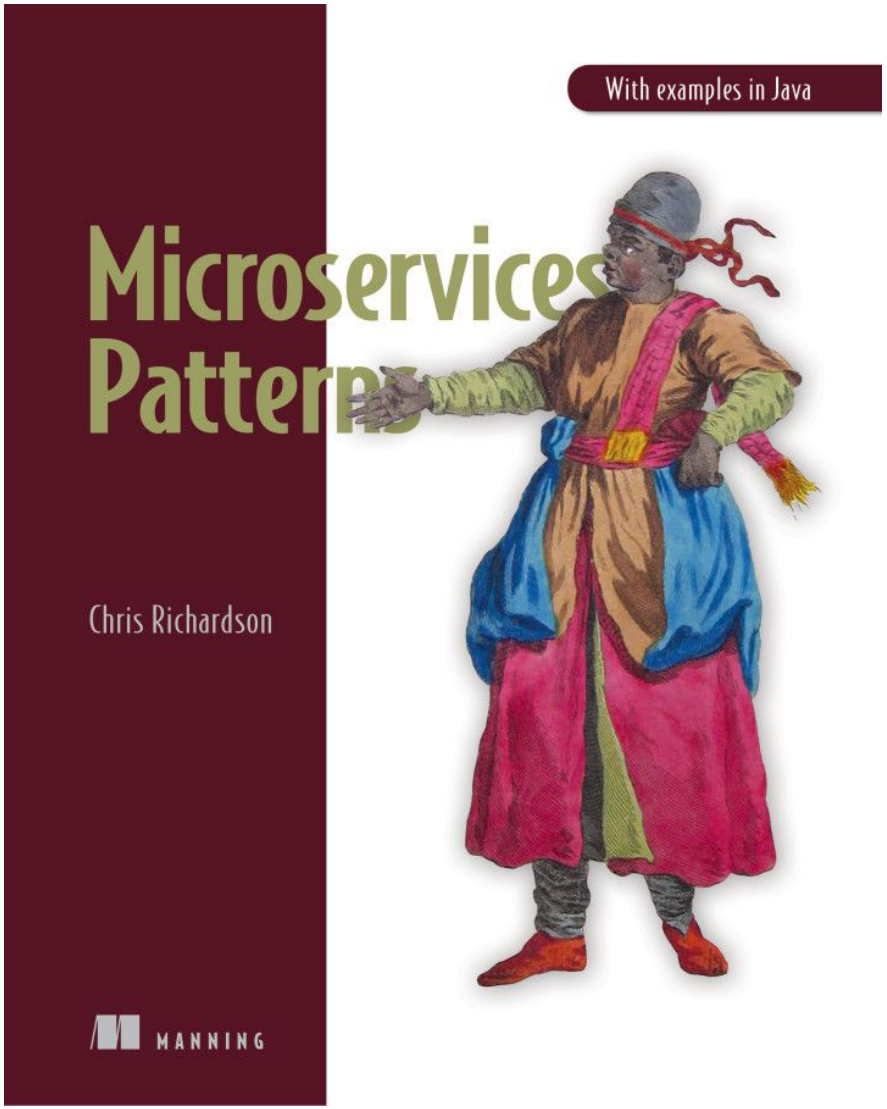
Agenda

Parte 2

- Arquitecturas monolíticas
- Arquitecturas de microservicios
  - Principios, ventajas, buenas prácticas
- Patrones asociados a microservicios
- ¿Cómo pasar de una arquitectura monolítica a microservicios?

Materiales de referencia



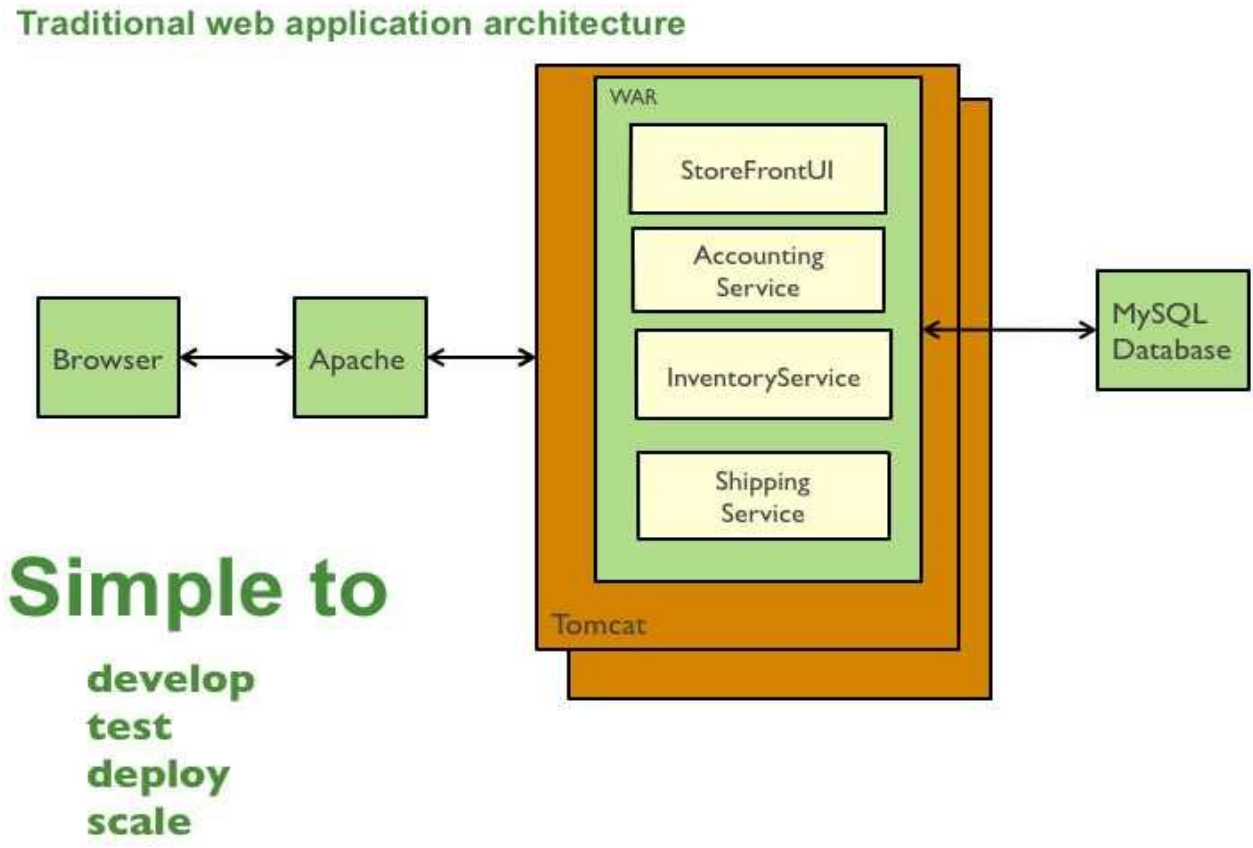


https://microservices.io/
https://microservices.io/book

Arquitecturas monolíticas

Arquitecturas monolíticas

- ➔ Un solo componente. Aplicación autocontenida
- ➔ “Simple” de:
  - Desarrollar
  - Escalar
  - Desplegar



microservices.io/patterns/monolithic.html

Problemas de las arquitecturas monolíticas

¿Qué pasa a medida que el proyecto crece?

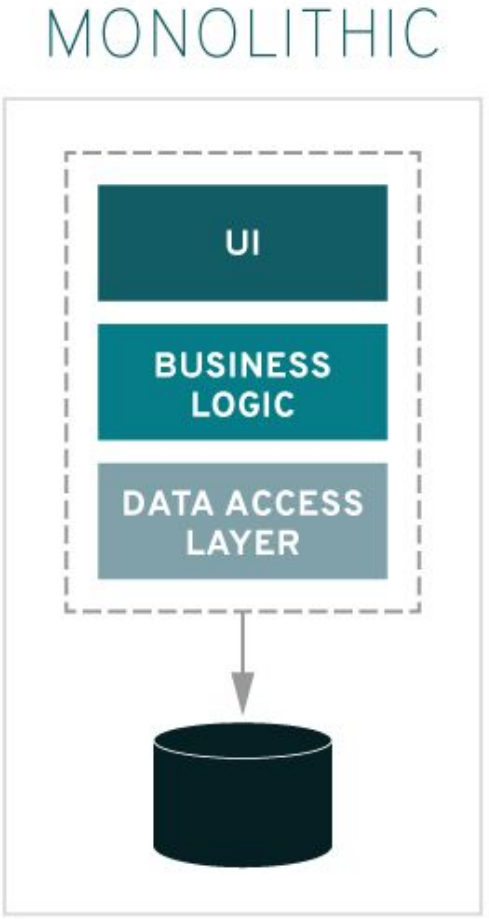
¿Y cuando cambien los requerimientos?

¿Y si cambian las cargas y requerimientos de recursos?

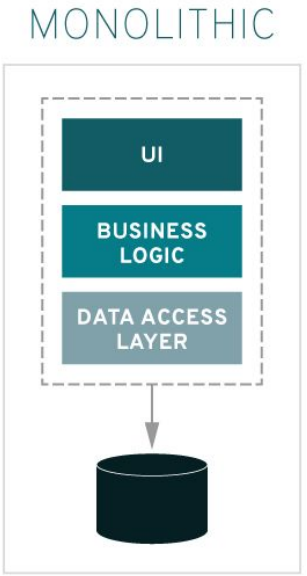
¿Y cuando falle algo?

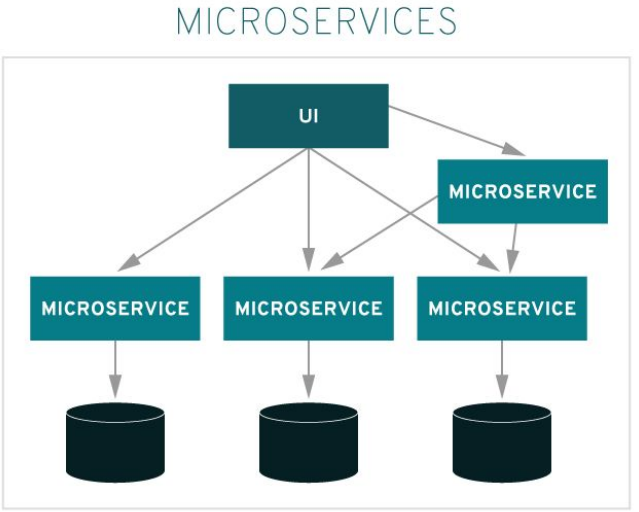
Problemas de las arquitecturas monolíticas

- ➔ Funcionalidades acopladas.
- ➔ Conforma un único punto de fallas
- ➔ Mucho código, difícil de entender y que sobrecarga IDEs.
- ➔ Impacto en el servicio ante modificaciones.
- ➔ Difícil de mantener el desarrollo en equipos.
- ➔ Cuando escala, escala todo el sistema.



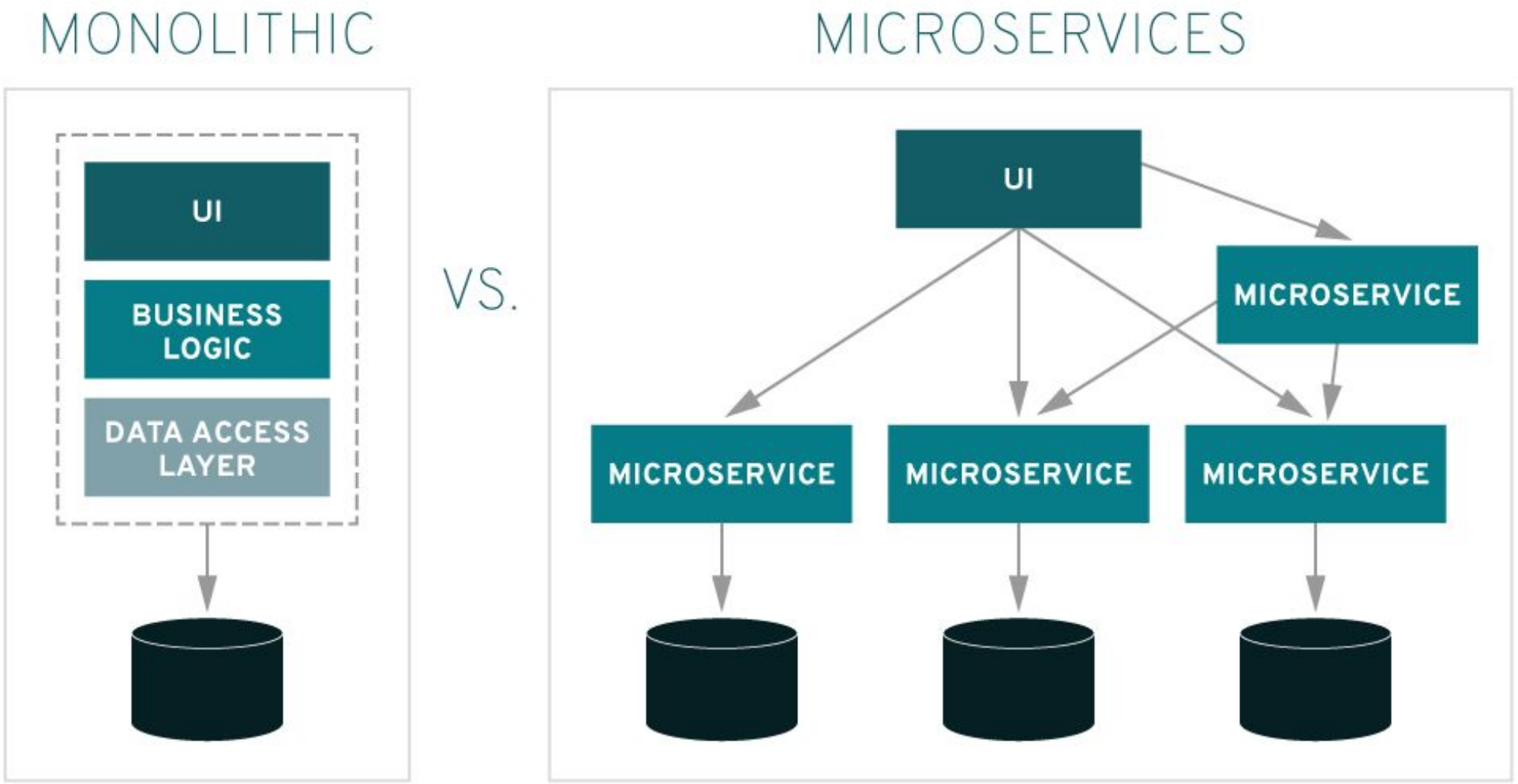
Arquitecturas monolíticas





Arquitecturas de microservicios

Arquitecturas de microservicios

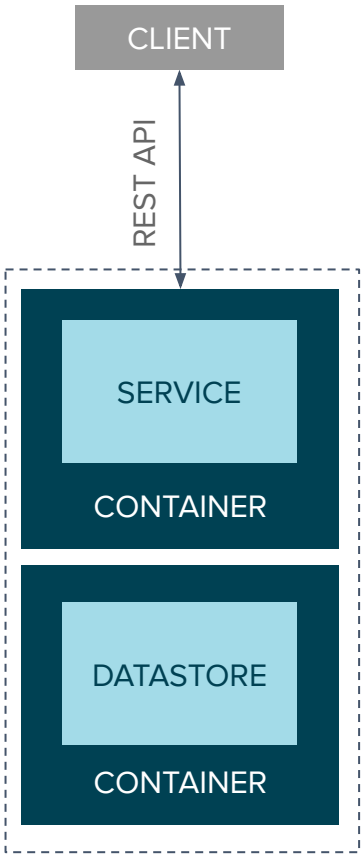


Según Martin Fowler..

"...describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data."

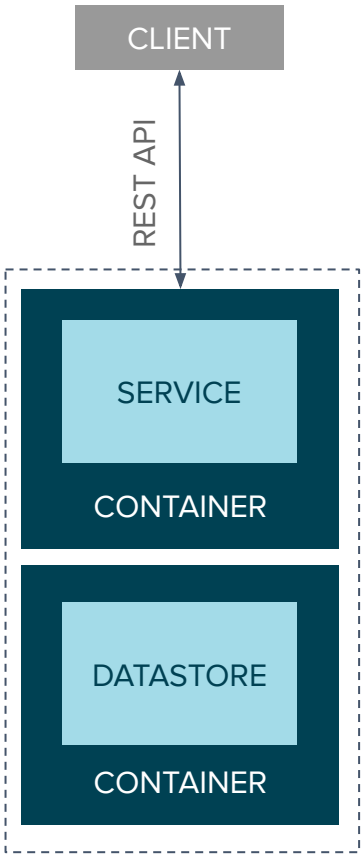
Martin Fowler, 2014,
(https://martinfowler.com/articles/microservices.html)

Los microservicios deben ser:



- ➔ Fáciles de mantener y probar.
- ➔ Independientes o tener poco acoplamiento entre ellos.
- ➔ Capaces de desplegarse independientemente.
- ➔ Capaces de ser desarrollados por un equipo pequeño.

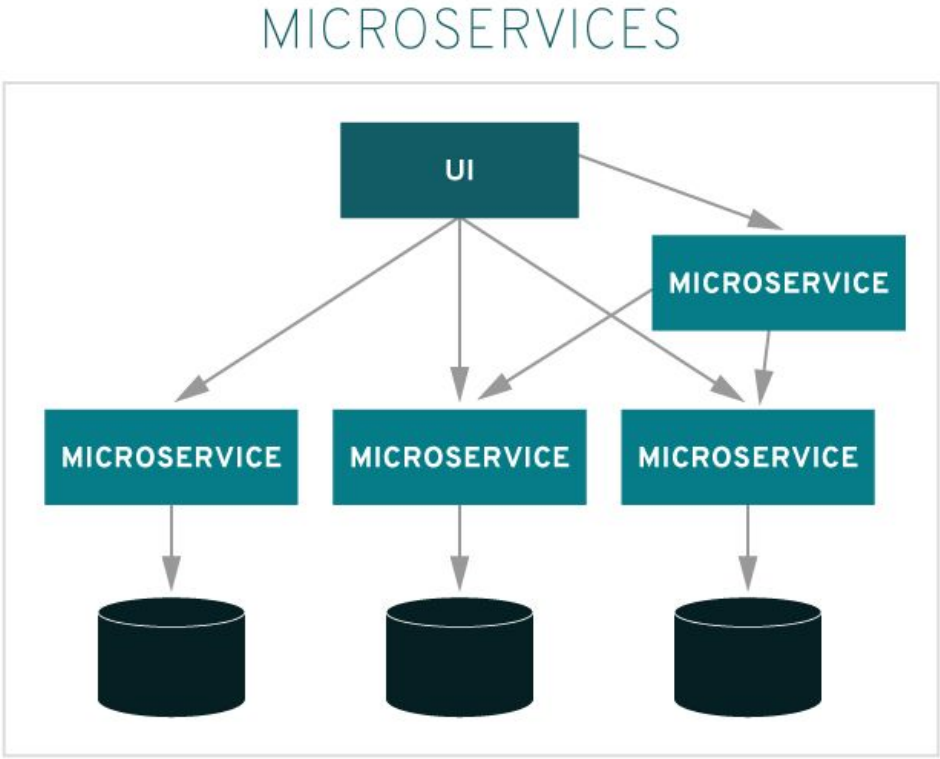
Algunos principios



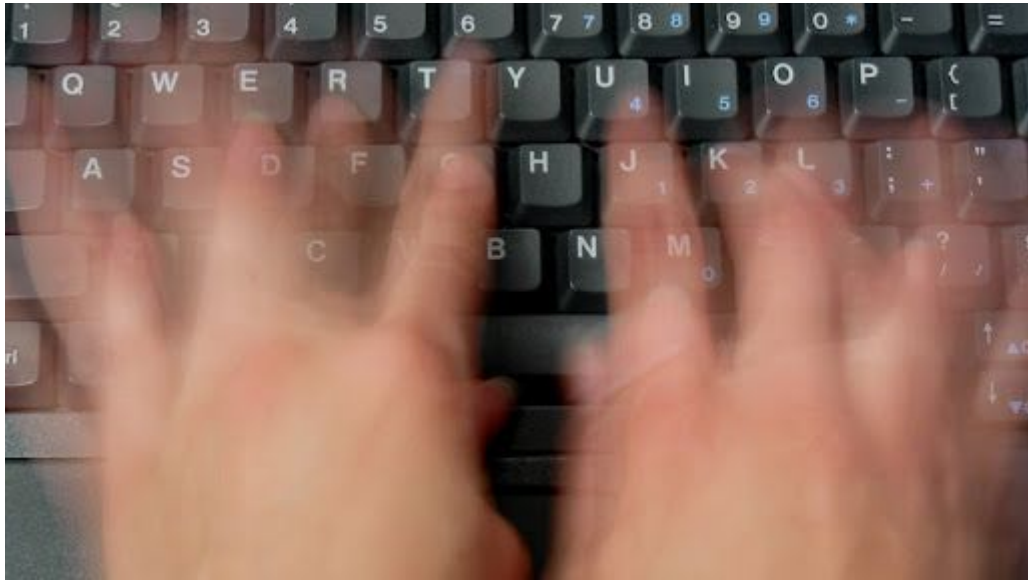
- ➔ DevOps, CI/CD.
- ➔ Se combinan para formar un sistema.
- ➔ Independencia: despliegue, liberación y escalamiento.
- ➔ Políglotas.
- ➔ Mecanismos de comunicación livianos. Por ej: HTTP.
- ➔ Buena definición de APIs.
- ➔ Descentralización de datos.

Arquitecturas de microservicios

- ➔ Puedo escalar solo lo necesario.
- ➔ Si un servicio falla, no necesariamente fallan los otros.
- ➔ Independencia entre servicios:
  - ◆ Escalamiento
  - ◆ Tecnologías
- ➔ Código más simple y de responsabilidad definida
- ➔ 1 equipo => 1 microservicio
- ➔ Disminución de impactos en servicio ante modificaciones.

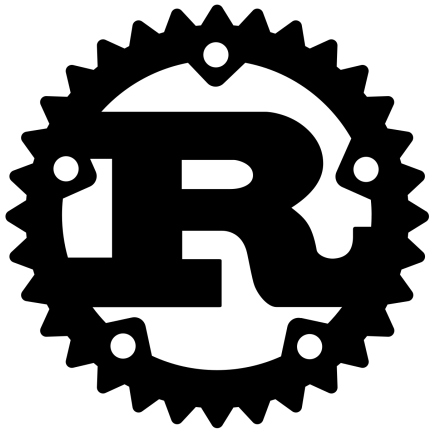


Fast time to market

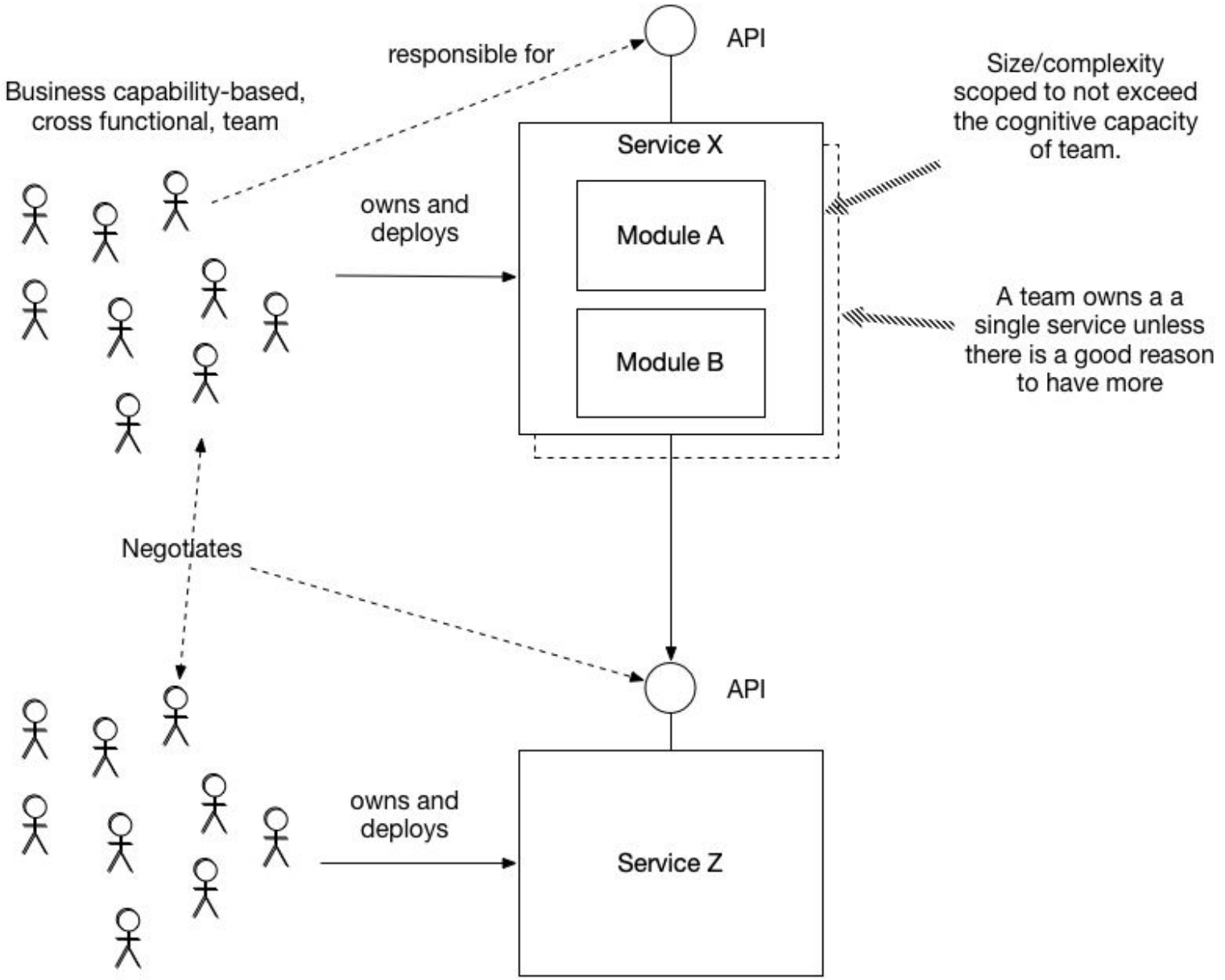


~~Don't fix what isn't broken.~~

Elegir la mejor tecnología en cada caso



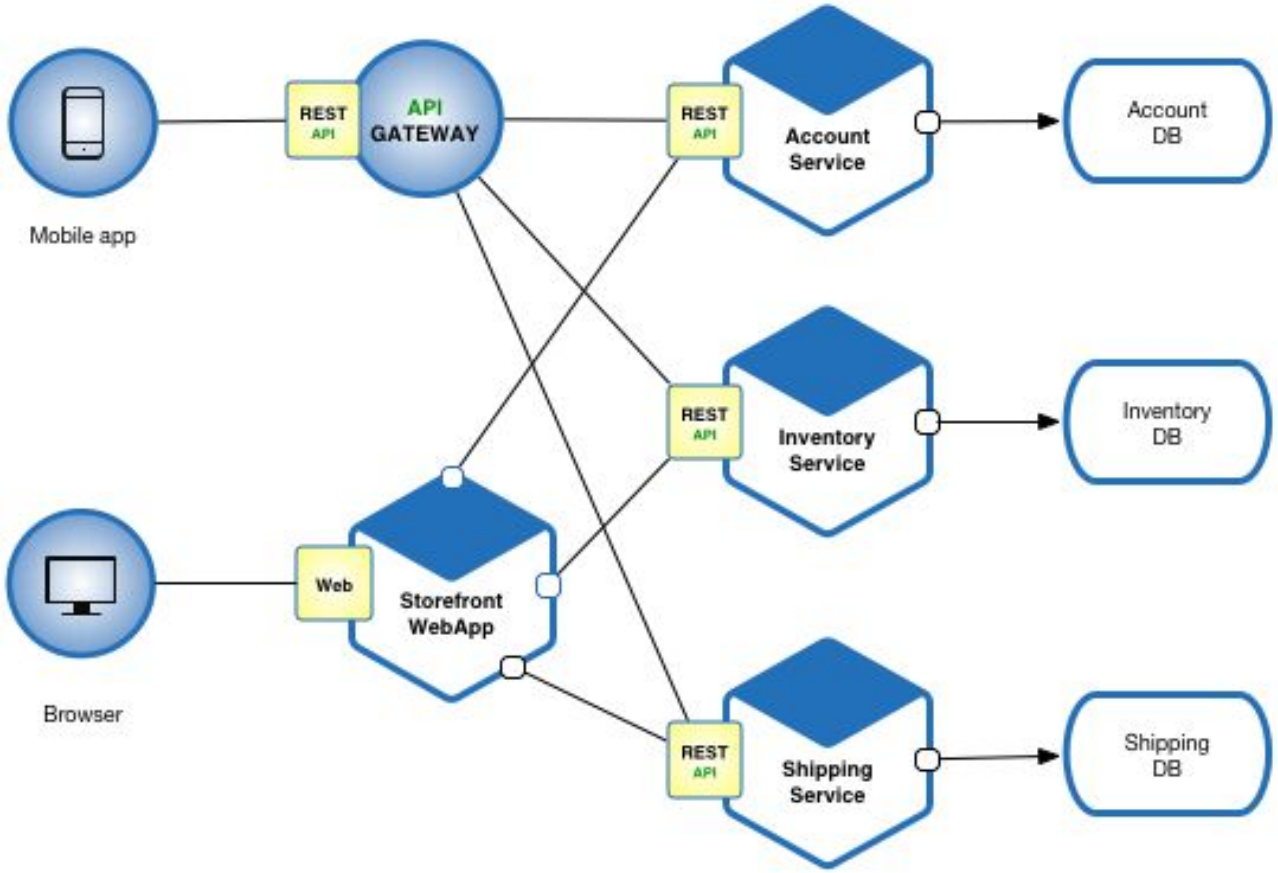
Un servicio por equipo



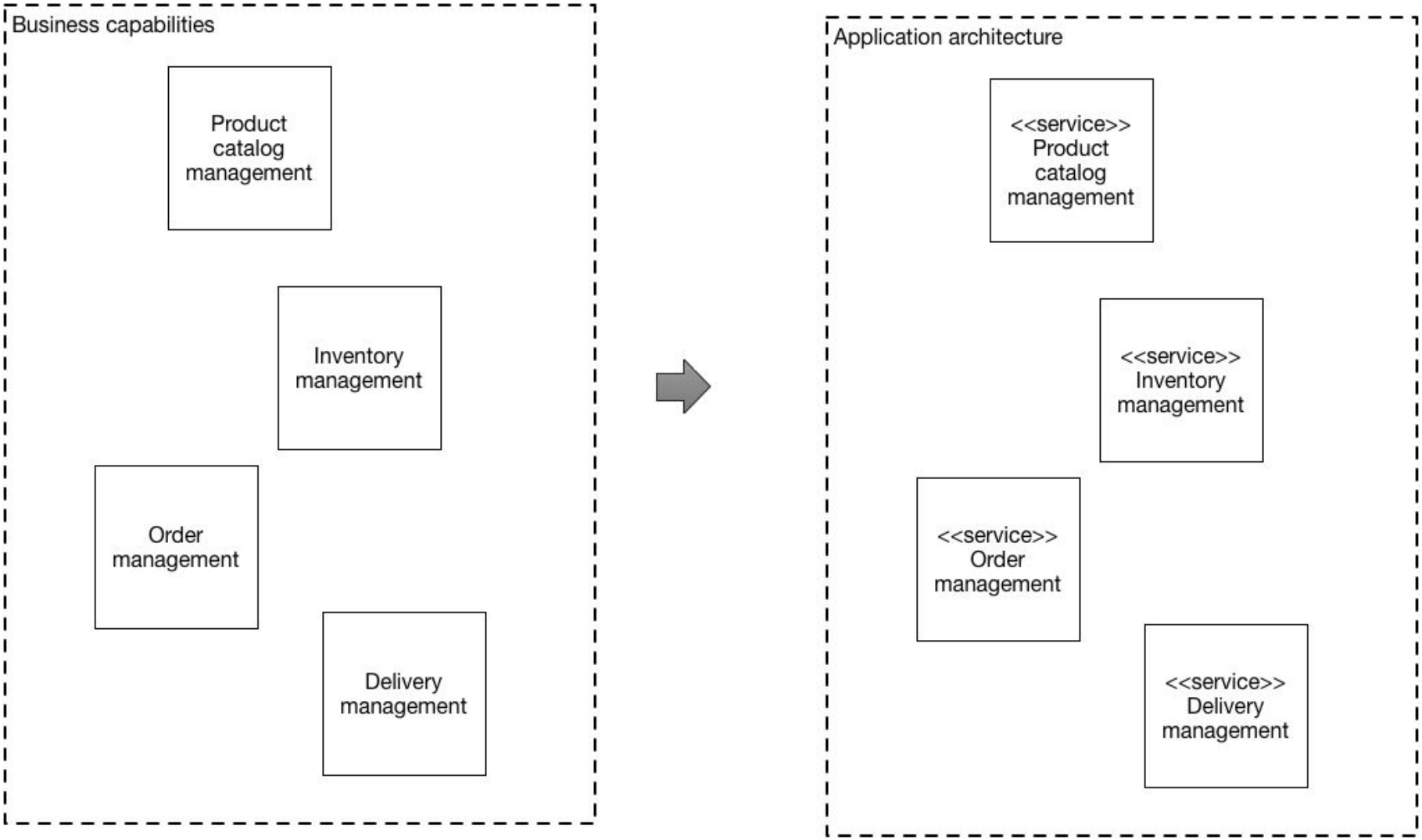
https://microservices.io/patterns/decomposition/service-per-team.html

¿Cómo definir los microservicios?

- ➔ Áreas/capacidades de negocio.
- ➔ Subdominios.

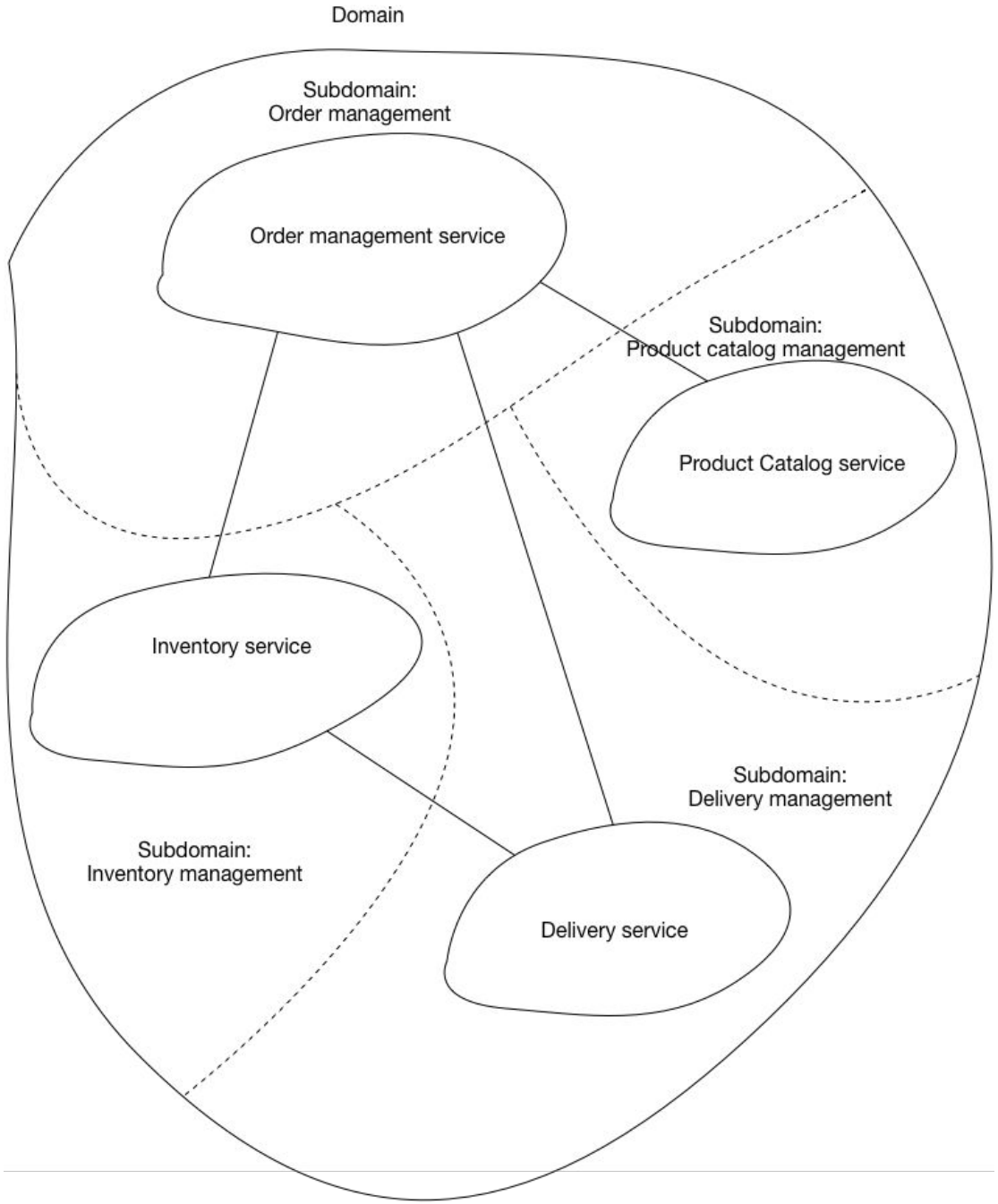


Por áreas/capacidades de negocio.



https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

Por subdominios

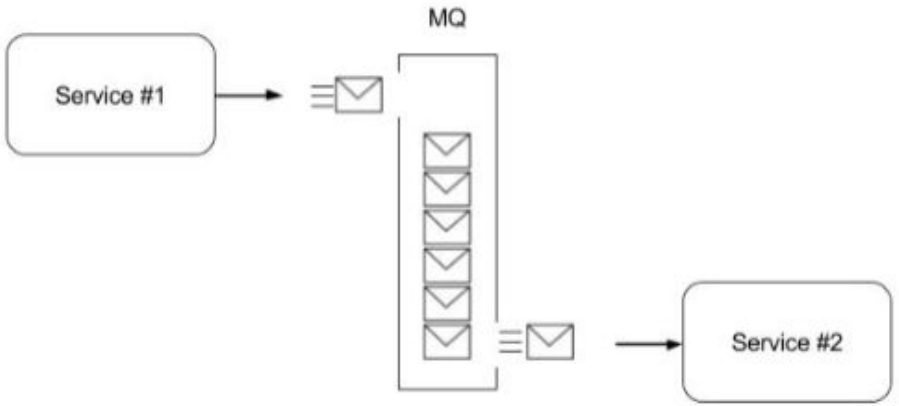


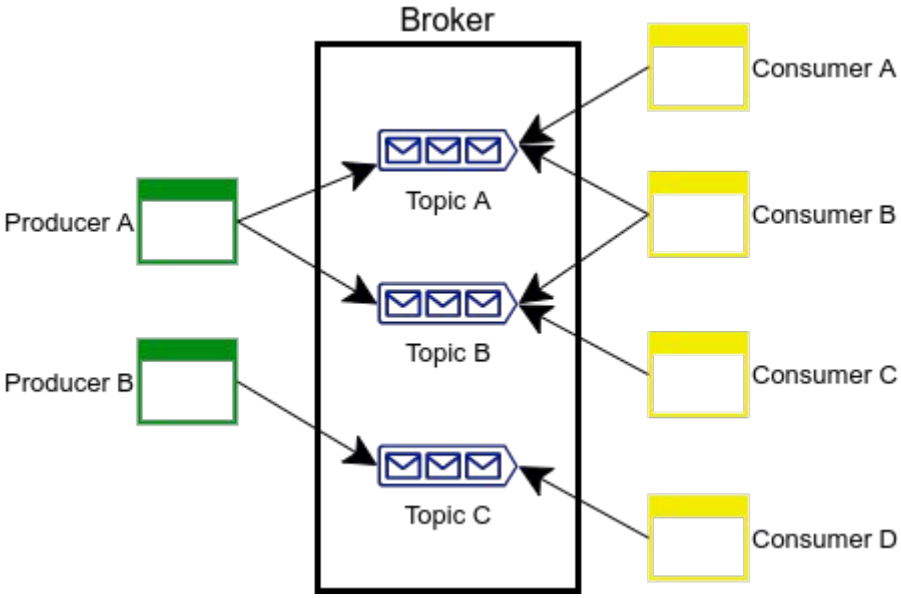
https://microservices.io/patterns/decomposition/decompose-by-subdomain.html

Patrones relacionados a microservicios

Comunicación interna: Mensajería

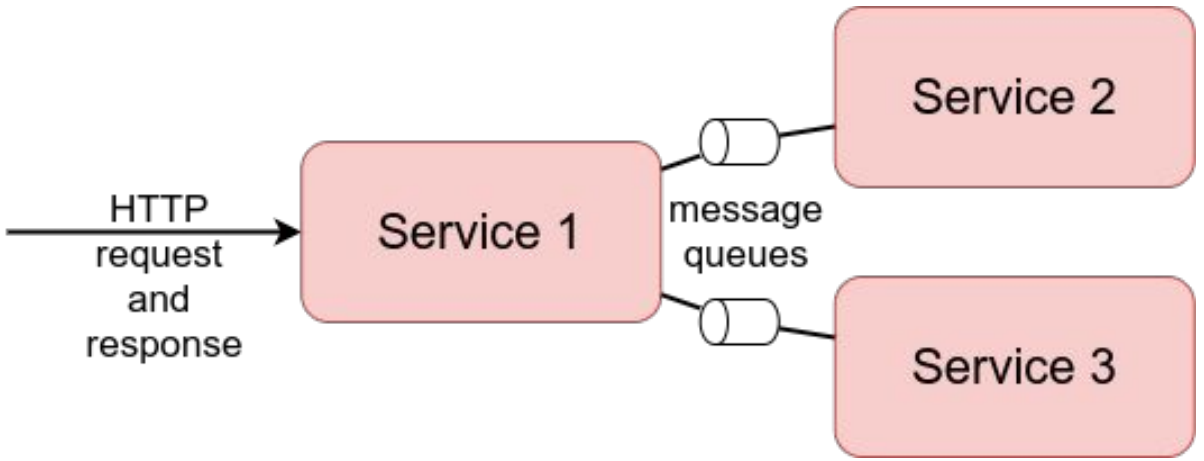
- ➔ La comunicación sincrónica produce bloqueos que pueden afectar performance y disponibilidad.
- ➔ Evitar llamadas bloqueantes para aumentar tiempos de respuesta y capacidad
- ➔ Usamos mensajería para evitar esos problemas y tener una comunicación asincrónica.



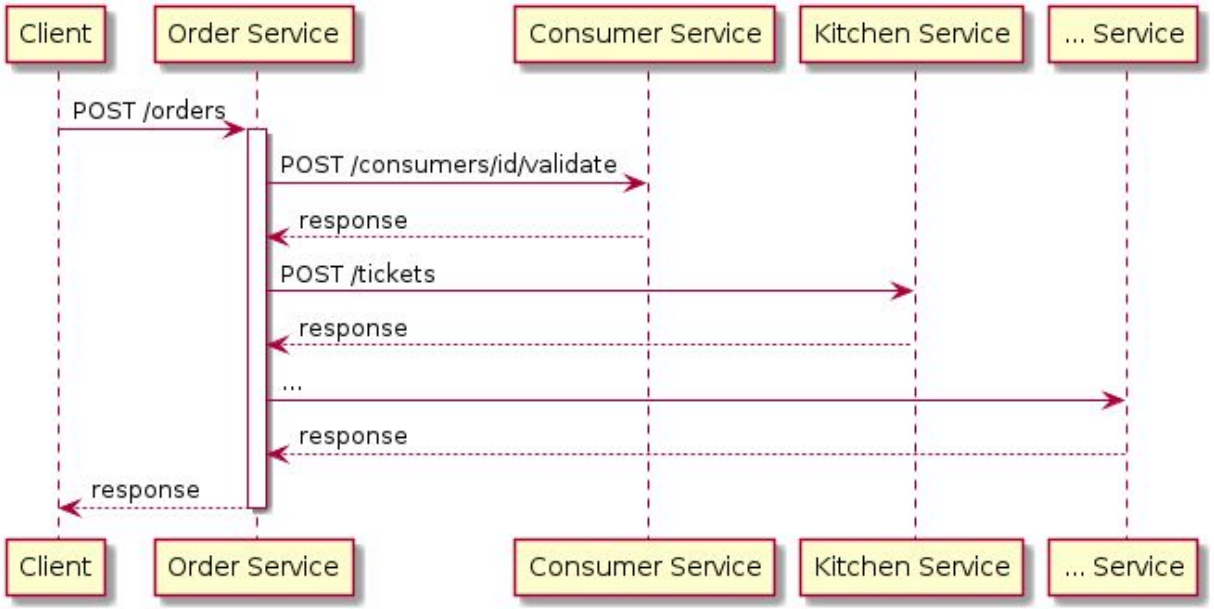


Self-contained service

- ➔ Usamos mensajería internamente, pero, ¿qué pasa si la comunicación desde el cliente externo es sincrónica?
- ➔ Utilizamos el patrón self-contained service

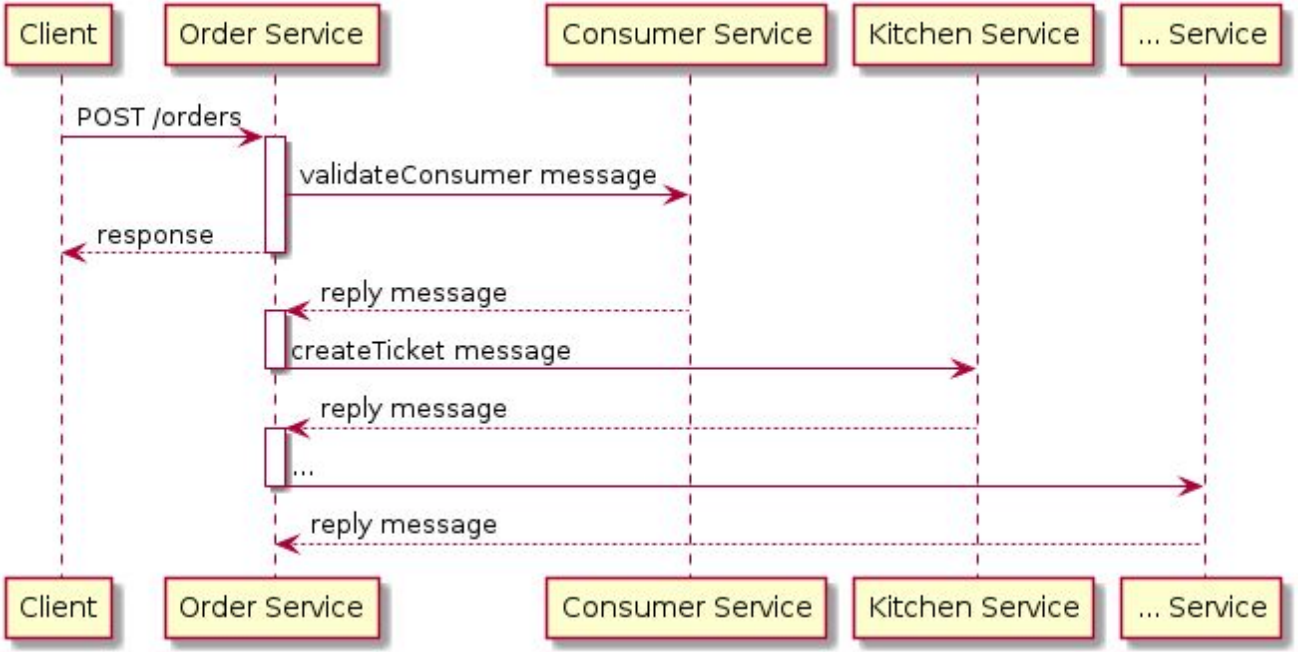


Self-contained service



Toda comunicación es sincrónica. Problemas!

Delegación asincrónica a los servicios internos



Configuración externa

Contexto y problema

- ➔ Los servicios usualmente tienen parámetros que cambian según el ambiente: dev, test, prod.
- ➔ Por ejemplo, conexión a bases de datos, a brokers de mensajería, a servicios de terceros, a servicios internos, etc.
- ➔ El servicio debe poder utilizarse en distintos ambientes sin modificarlo.

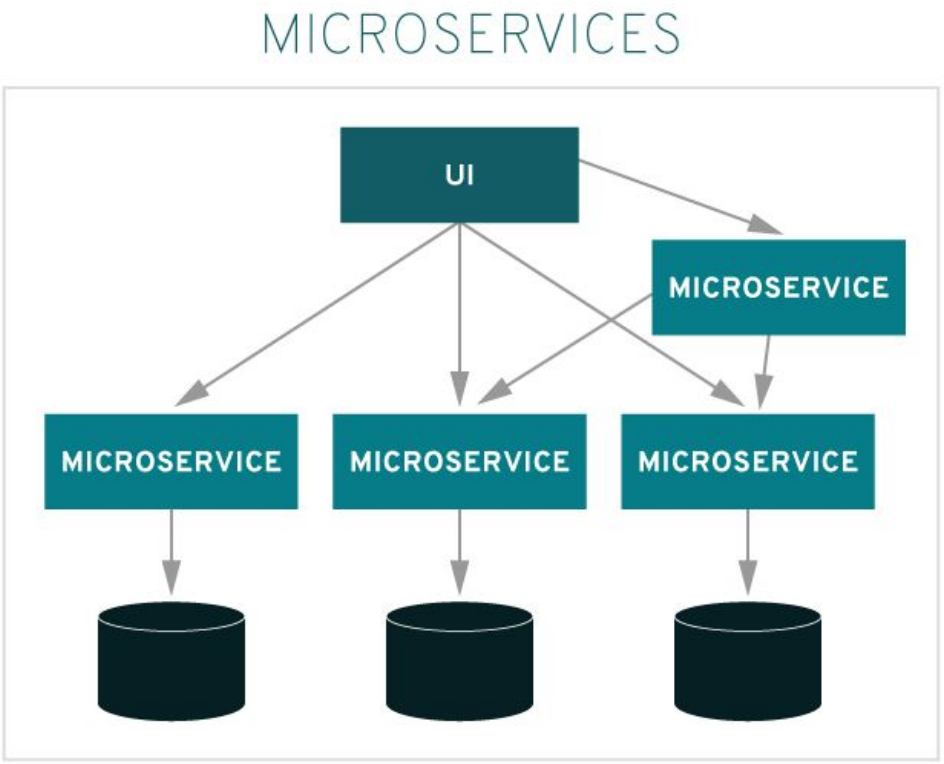
Solución

- ➔ Utilizamos configuración externa mediante variables de entorno o archivos de configuración

```
java -jar application.jar $DB_HOST $DB_USER $DB_PASS
```

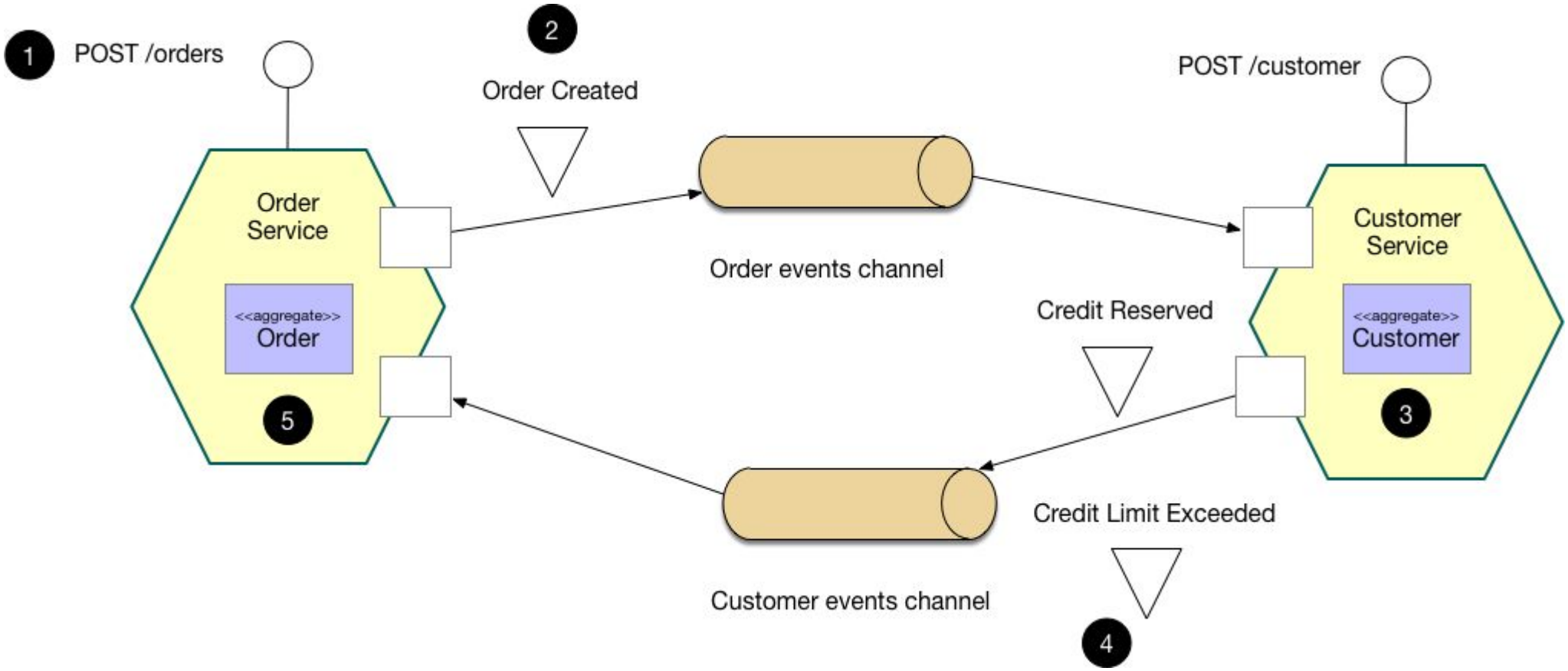
```
java -jar application.jar $CONFIG_PATH
```

Una base de datos por servicio



- ➔ Los microservicios deben estar desacoplados para desarrollarse, desplegarse y escalar de forma independiente.
- ➔ Cada microservicio tiene sus datos privados y son accesibles únicamente mediante su API.

¿Cómo mantener la consistencia de los datos?
Patrón Saga / Eventos



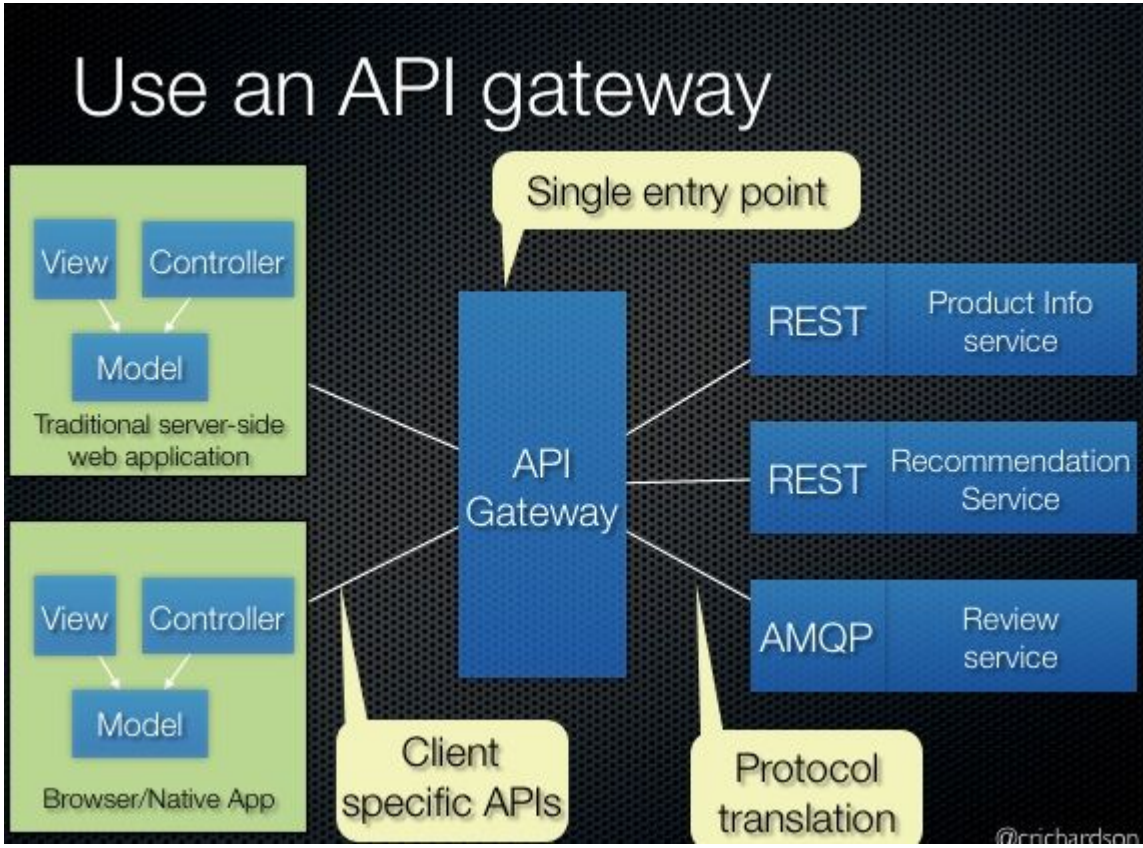
Test por servicio

- ➔ Testear microservicios puede ser complejo. Se comunican con otros servicios.
- ➔ ¿Cómo hacemos pruebas automatizadas en microservicios?

- ➔ Creamos conjuntos de test que funcionan aislados.
  - ◆ Utilizamos “test doubles” para los servicios que invoca.

API Gateway

- El sistema de microservicios provee funcionalidades que necesitan ser accedidas externamente.
- Esas funcionalidades las resuelven distintos microservicios.
- Se utiliza un API Gateway como único punto de entrada para los clientes



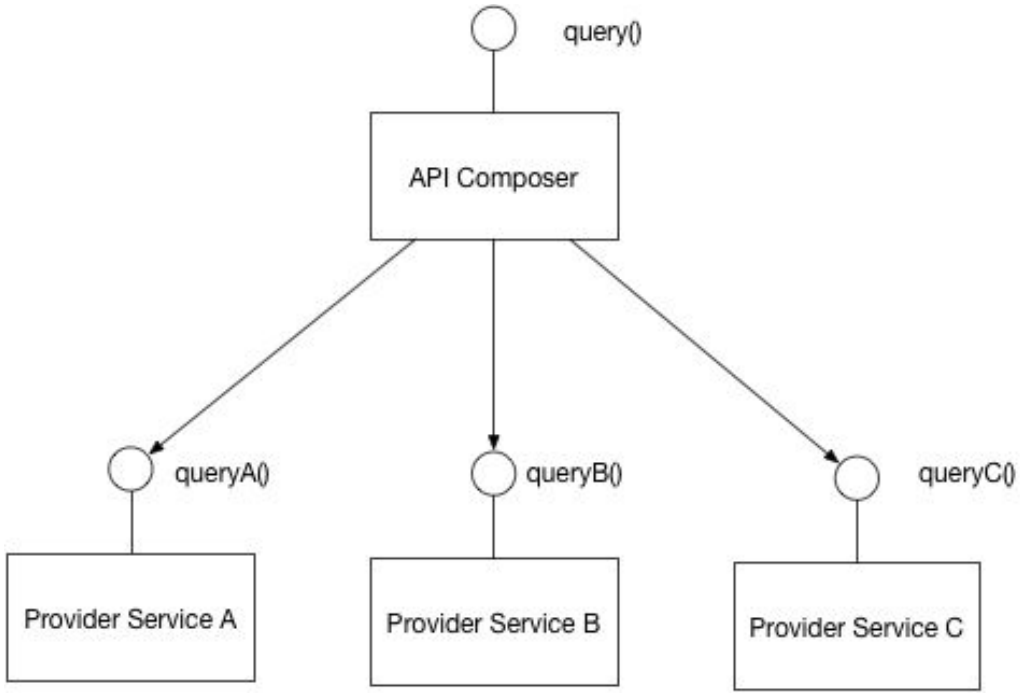
API Composition

Contexto y problema:

- ➔ Implementamos el patrón “una base de datos por servicio”.
- ➔ ¿Cómo consultamos datos de distintos microservicios?

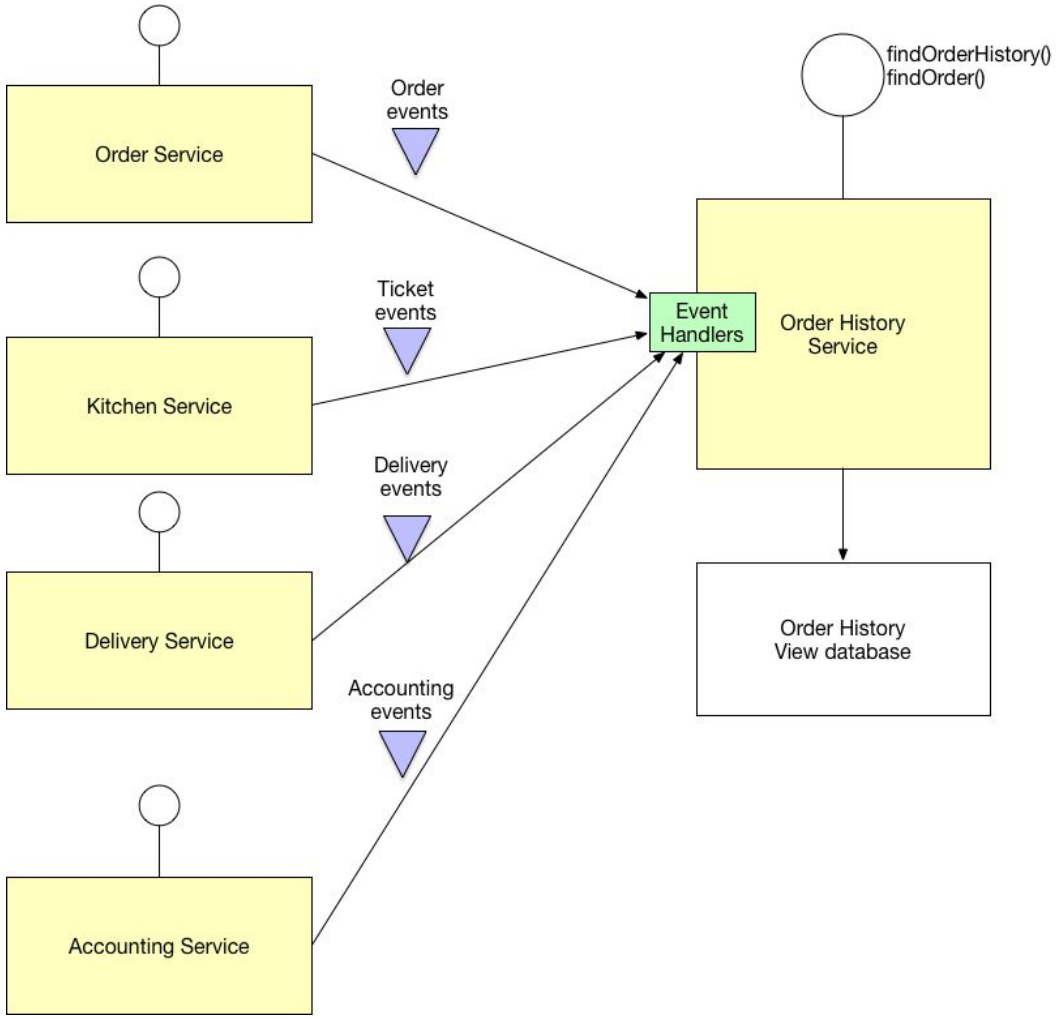
Solución:

- ➔ Utilizamos el patrón API Composition.



Command Query Responsibility Segregation (CQRS)

- Si bien se usa en otros contextos, lo podemos ver como una alternativa a API Composition.
- Crea una base de datos read-only de réplica diseñada para la consulta de datos deseada.
- La carga con eventos de los otros microservicios

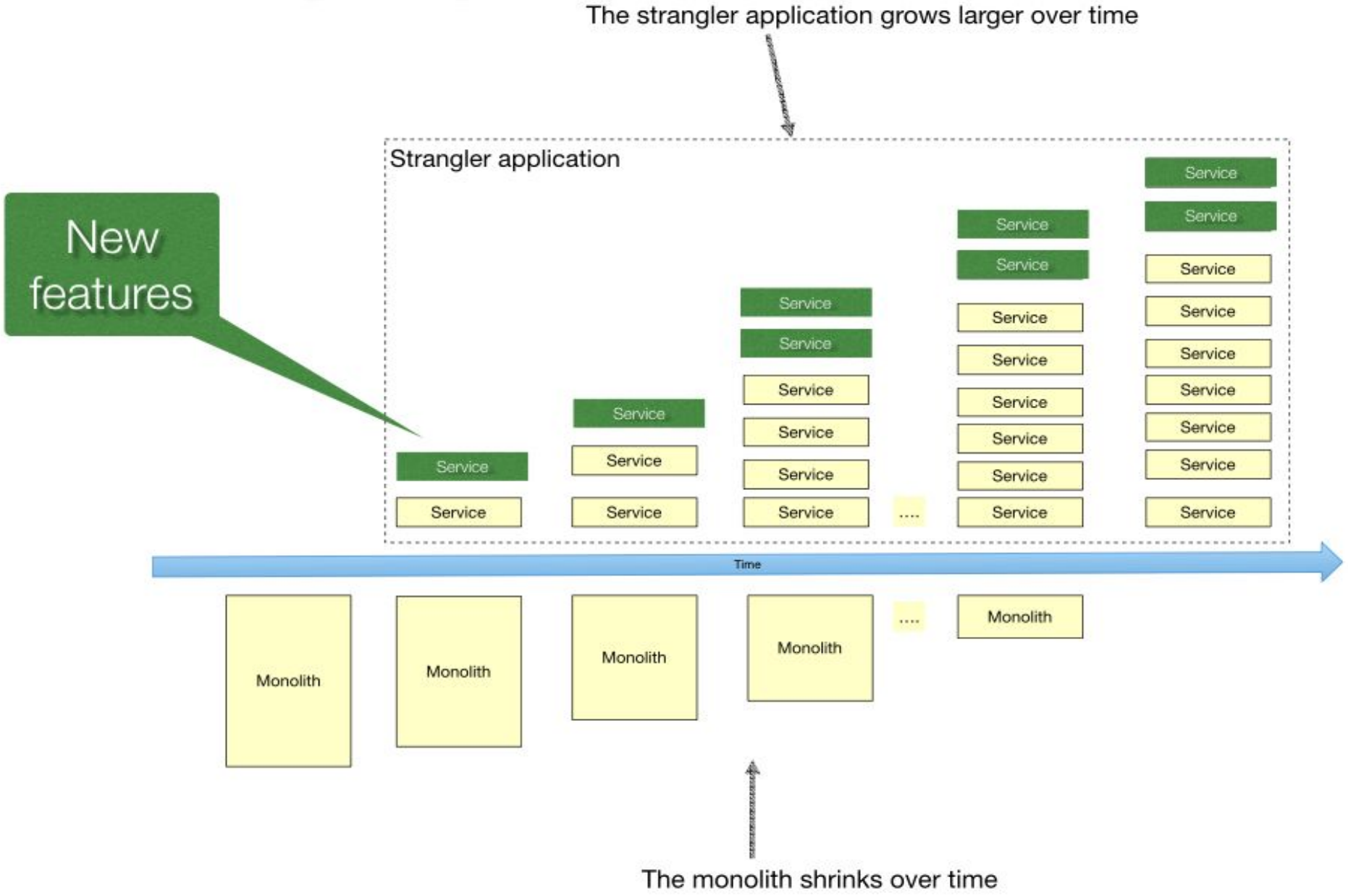


Monitoreo: agregación de logs, health check APIs y métricas.

- ➔ Centralizar Logs. Por ejemplo: ELK, AWS Cloudwatch.
- ➔ API de health check en cada servicio. Por ejemplo, `HTTP GET /health`
- ➔ Generar métricas y centralizarlas. Formatos push o pull.
  - ◆ Un servicio recomendado: Prometheus.

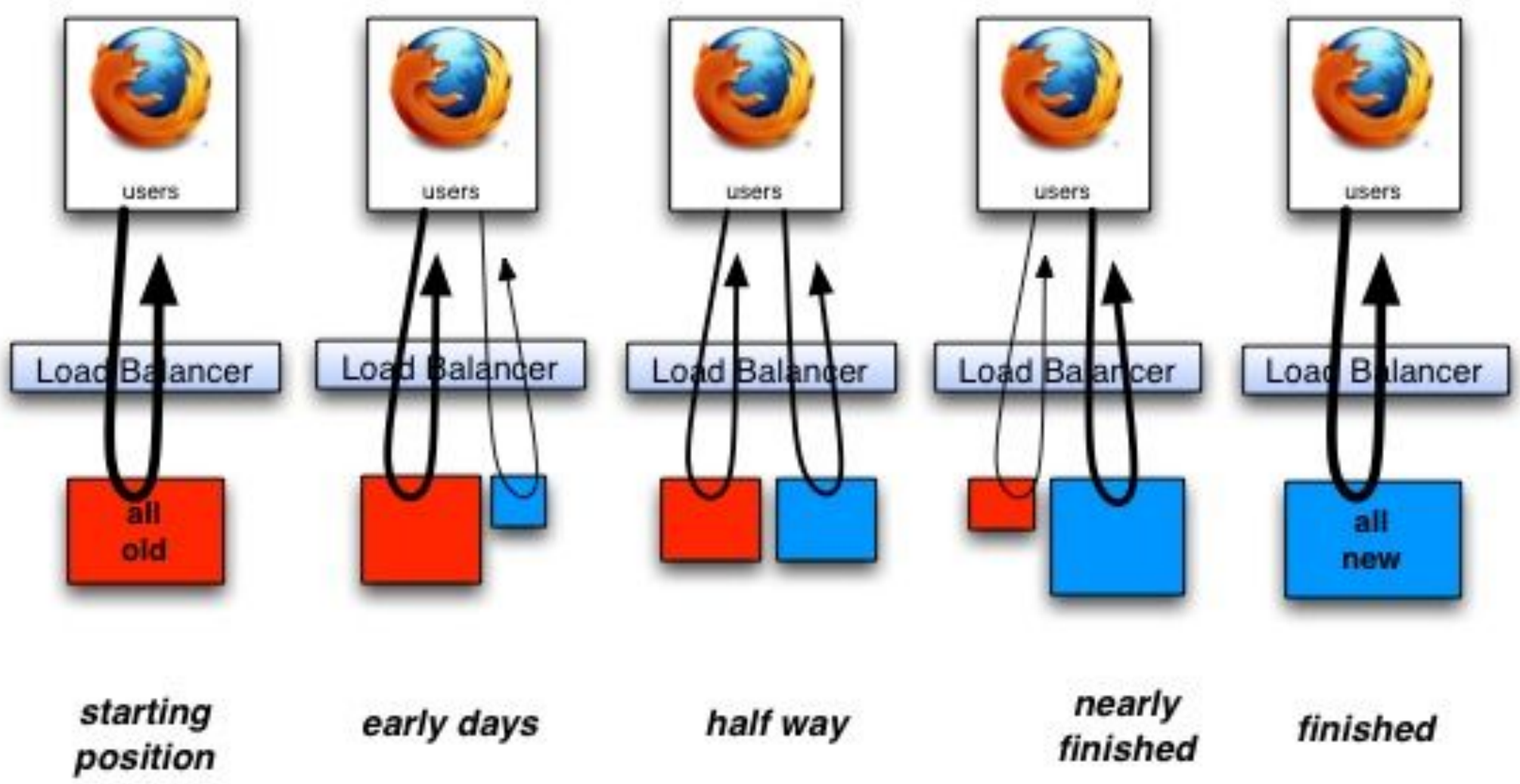
¿Cómo pasar de una arquitectura monolítica a microservicios?

Strangling the monolith



https://microservices.io/patterns/refactoring/strangler-application.html

Strangling the monolith



http://redhatgov.io/workshops/strangling_the_monolith/lab2/

Taller:
De monolito a microservicios