# Microservices

*Master 2*

# Monolithes vs Microservices

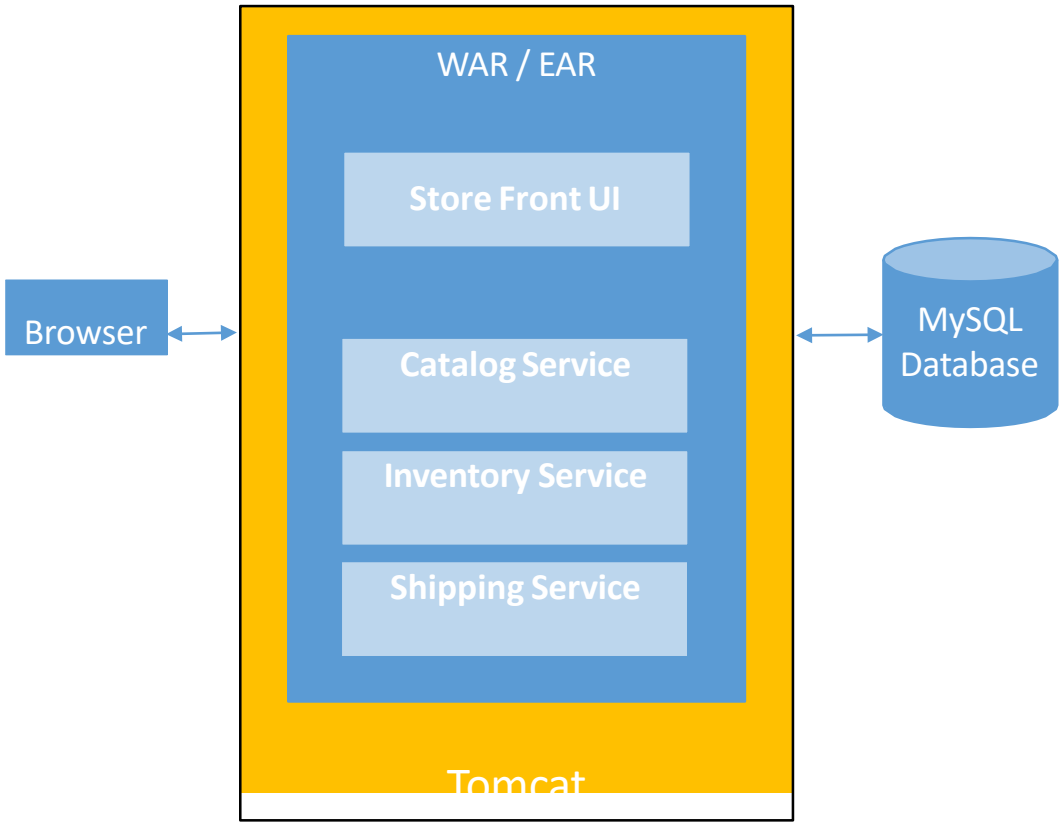
Monolithe

Microservices

No silver bullet

# Il était une fois … un Monolithe magnifique

# Monolithe: Définition



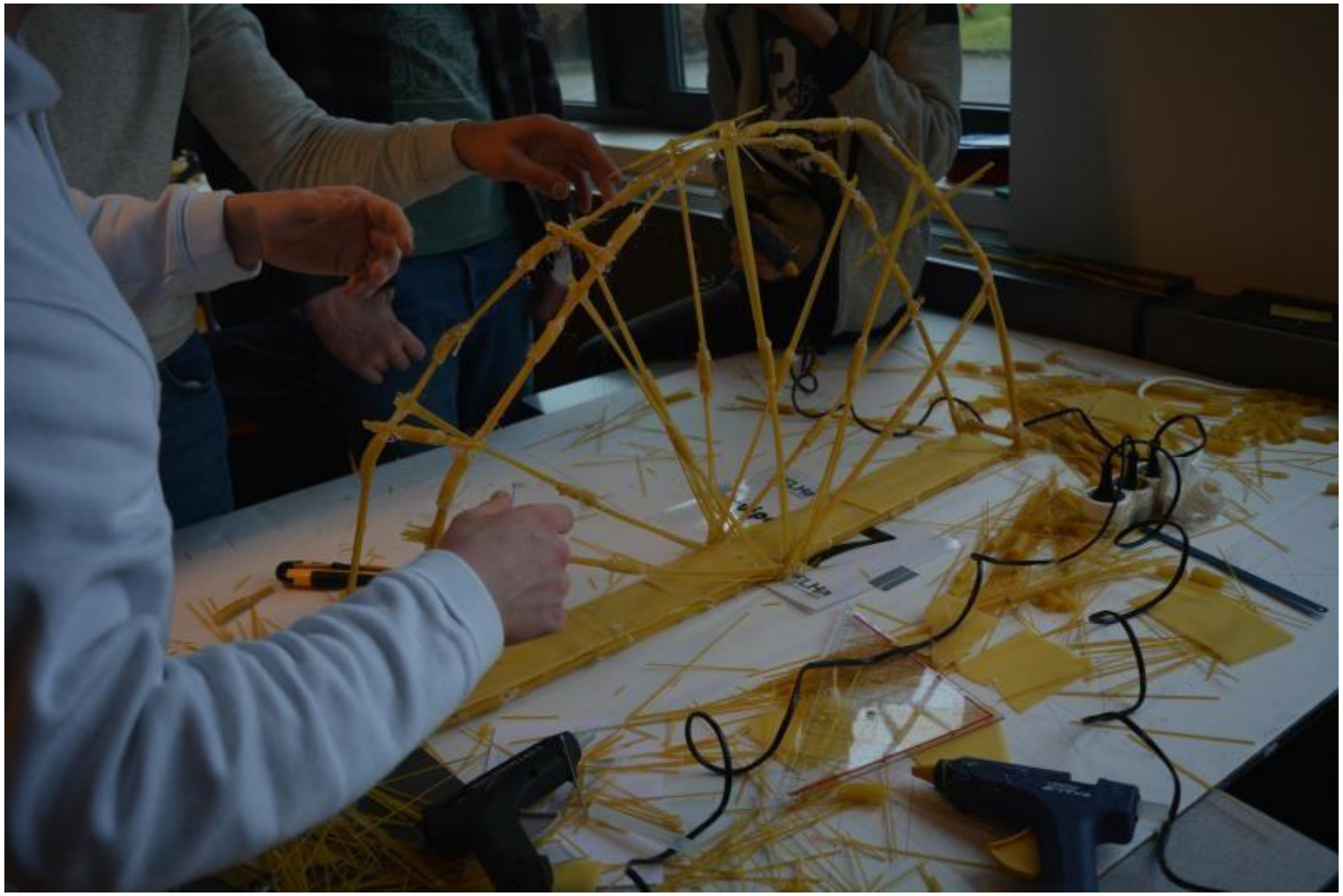

- Une seule unité d'exécution
- Composants fortement couplés
- Cadence de release long: > 3 mois
- Equipe assez grande > 10 pers

# Monolithes: Problèmes

# Collaboration difficile

## Quand l’application grossit … plusieurs équipes se marchent sur les pieds

# Testabilité lente et douloureuse



# Déploiement long et risqué

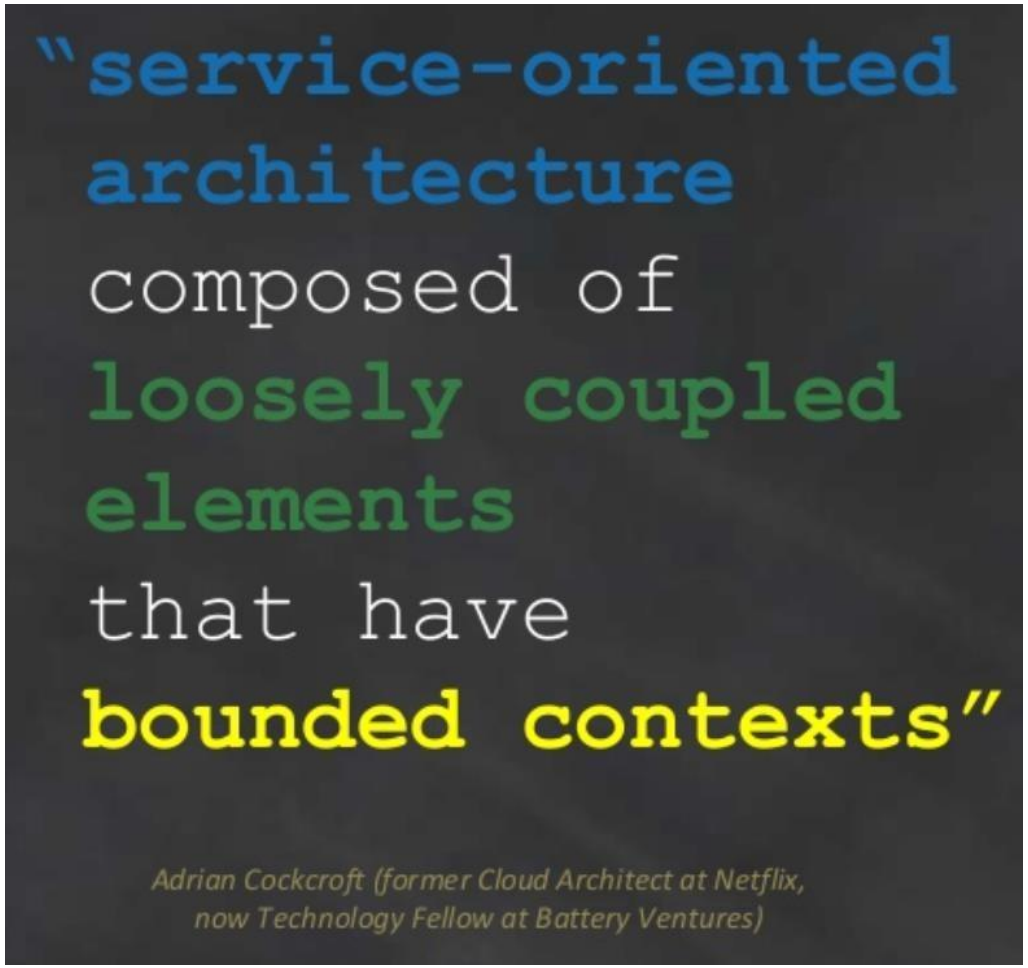
# Innovation rendue difficile

# Difficilement scalable

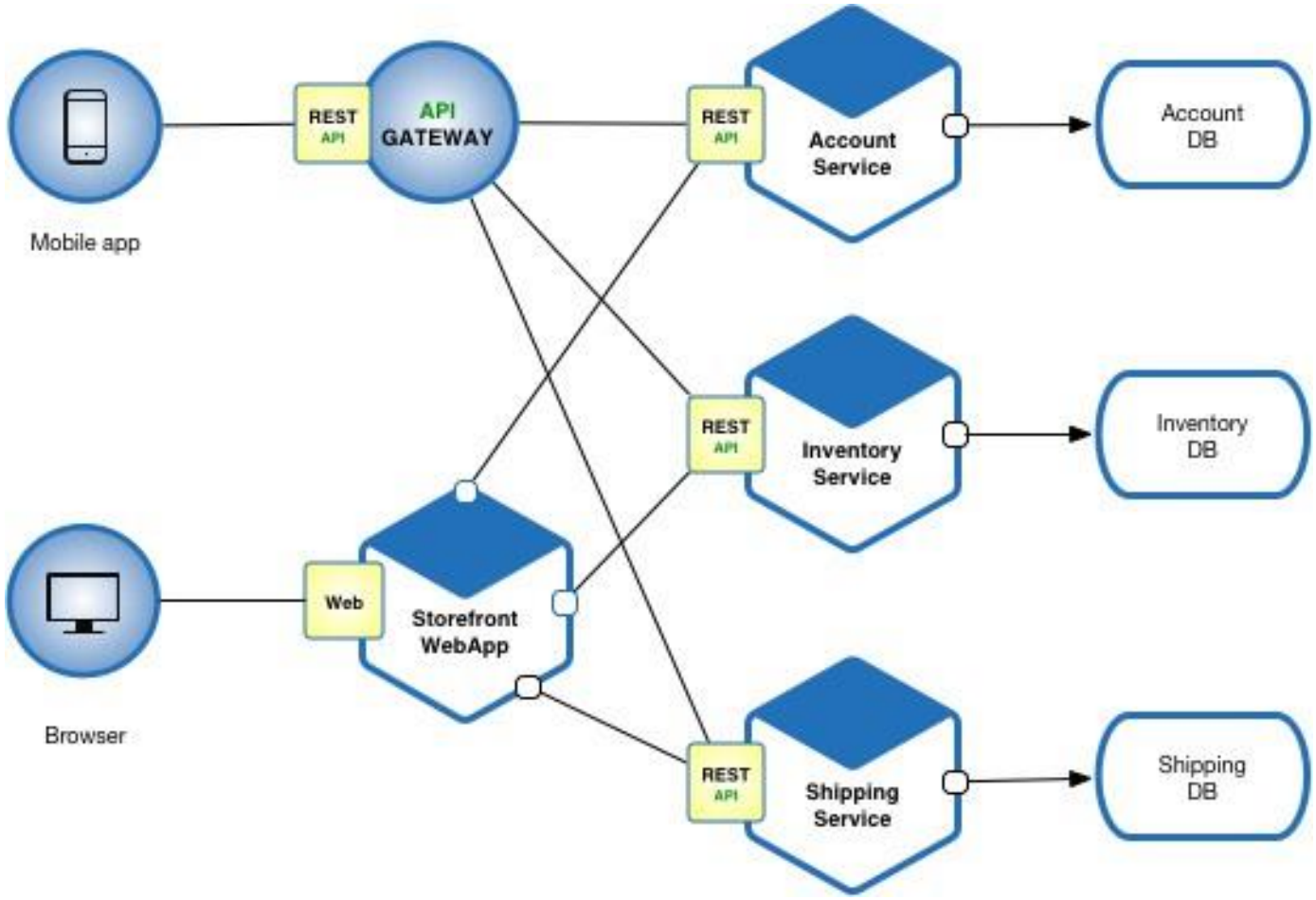
# Monolithe: Peut être une bonne solution

- Approche Monolithe First
- Solution parfois la plus adaptée:
  - équipe reste petite
  - time to market
  - clean code
  - livraison rapide

# Architecture microservices: Définition



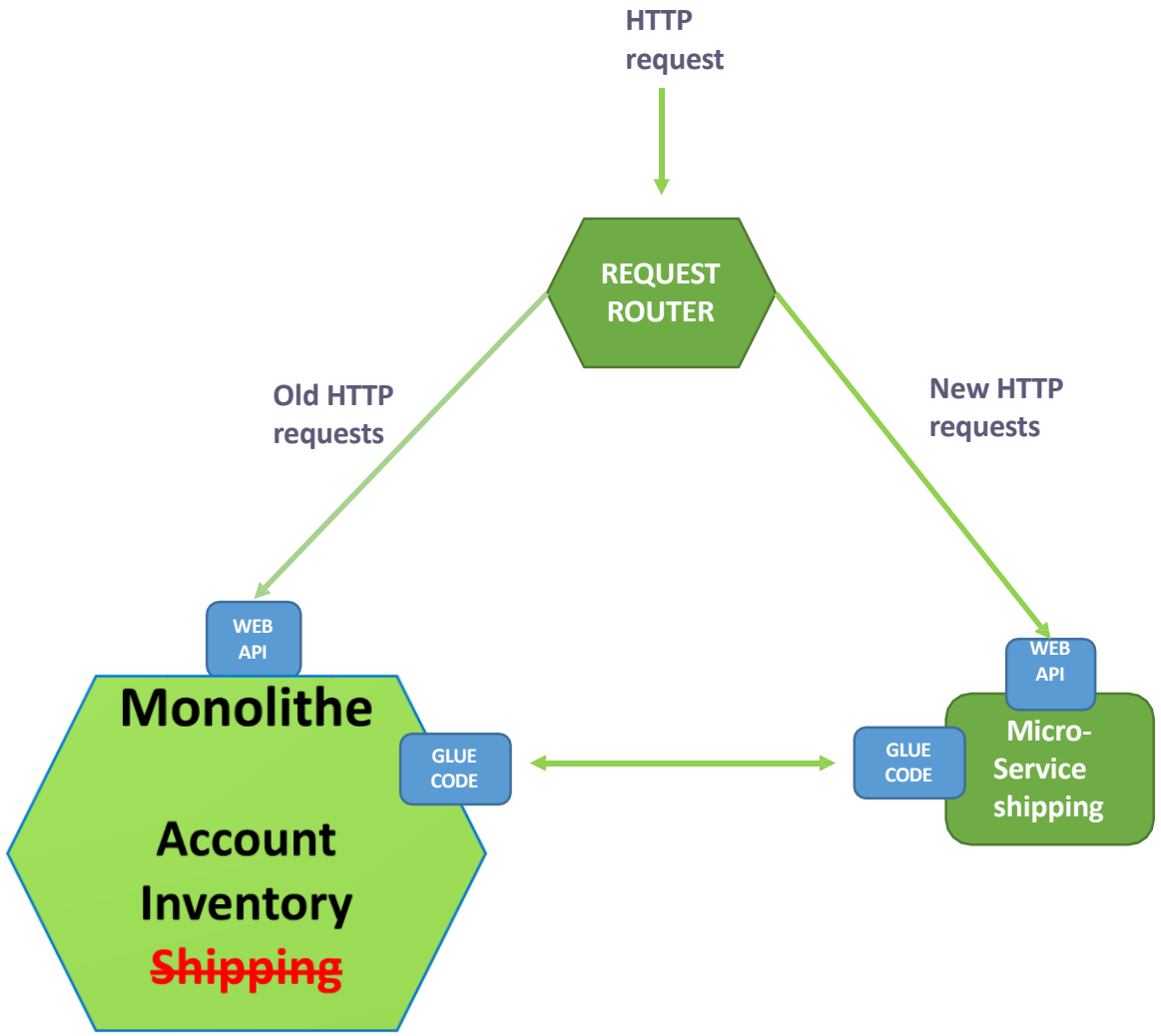
# Architecture microservices: Exemple



# Patterns de migration: Big Bang

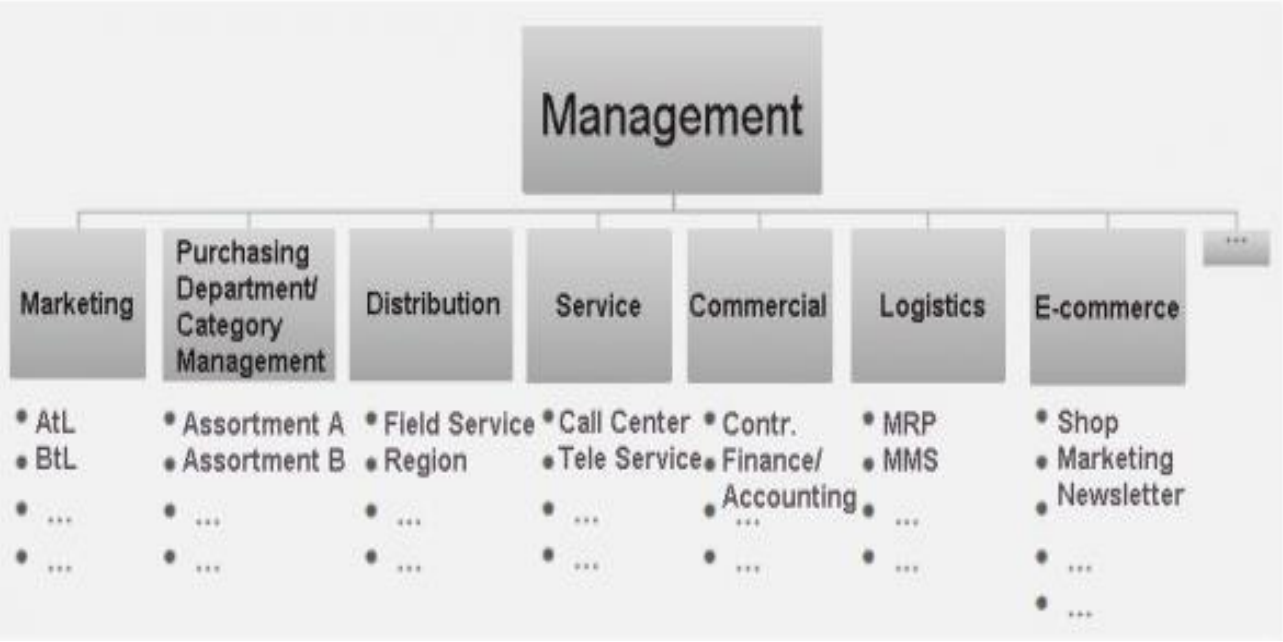
- Très long à développer
- L'existant bouge en même temps
- Plus compliqué et risqué !

# Patterns de migration: Strangler Pattern



# Comment découper les microservices ?

# Découper par département métier




- Décomposer en sous-domaines
- Un bon point de départ
- … mais assez limitant et peu créer des dépendances

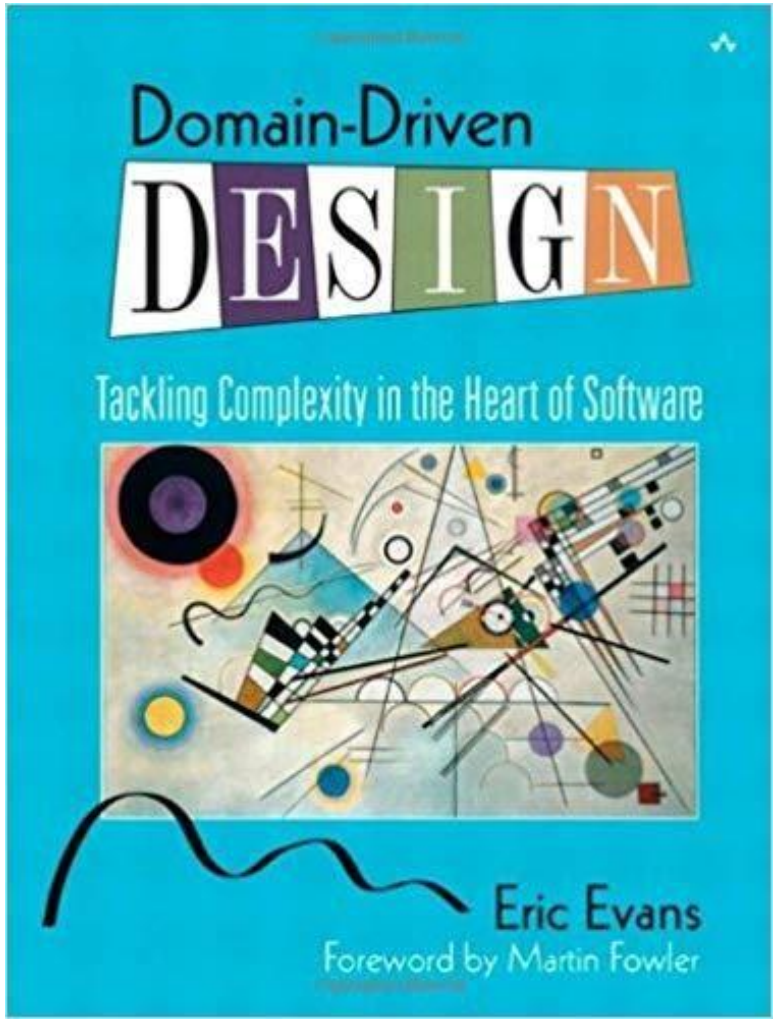
# Découper par objectif  (verbe d'action)

- Chaque domaine a un objectif
- Faire des interviews avec tous les acteurs (définir des patates)
- Exemple: Shipping, Billing, Recommandation, Navigation, Searching

# Découper selon une contrainte technique forte

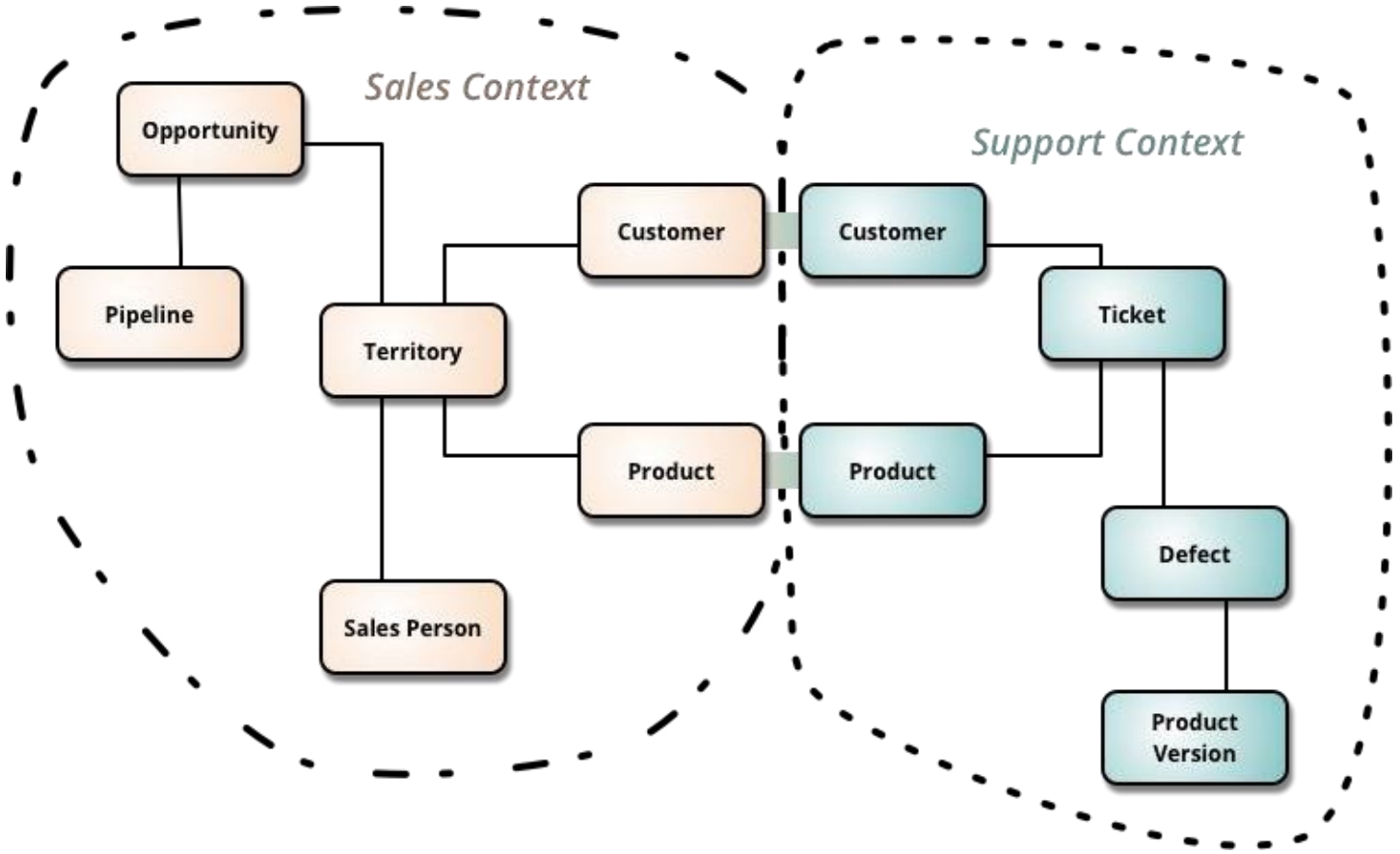
- Haute volumétrie vs faible volumétrie (BD NoSQL/Relationelle)
- Focus Lecture vs Ecriture (CQRS)
- Calcul rapide vs exact (finance de marché)

# Découper par Bounded context (DDD)



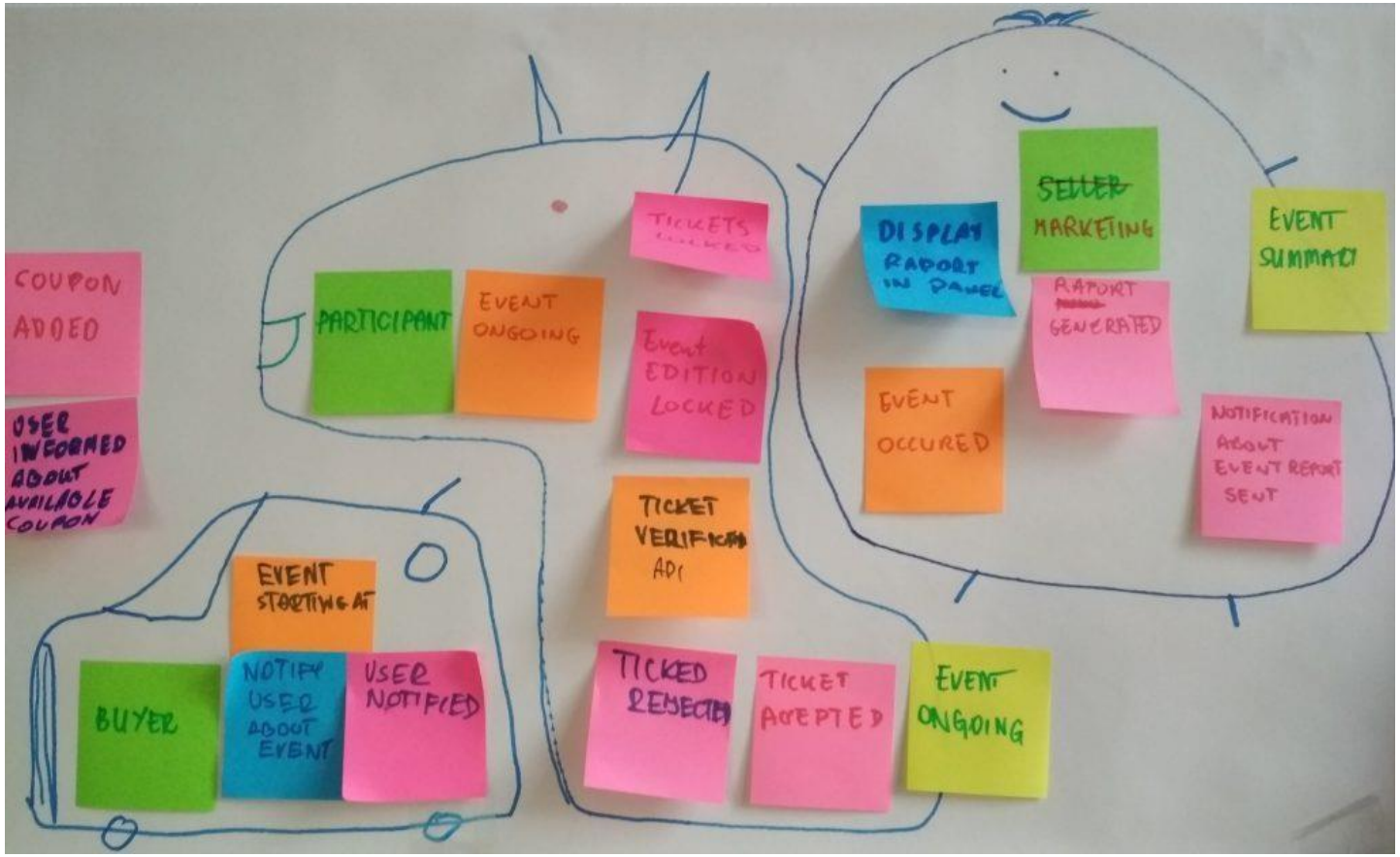

- Ubiquitous langage
- Bounded Context
- Aggregate root
- Repository
- Domain Service

# Découper par Bounded Context

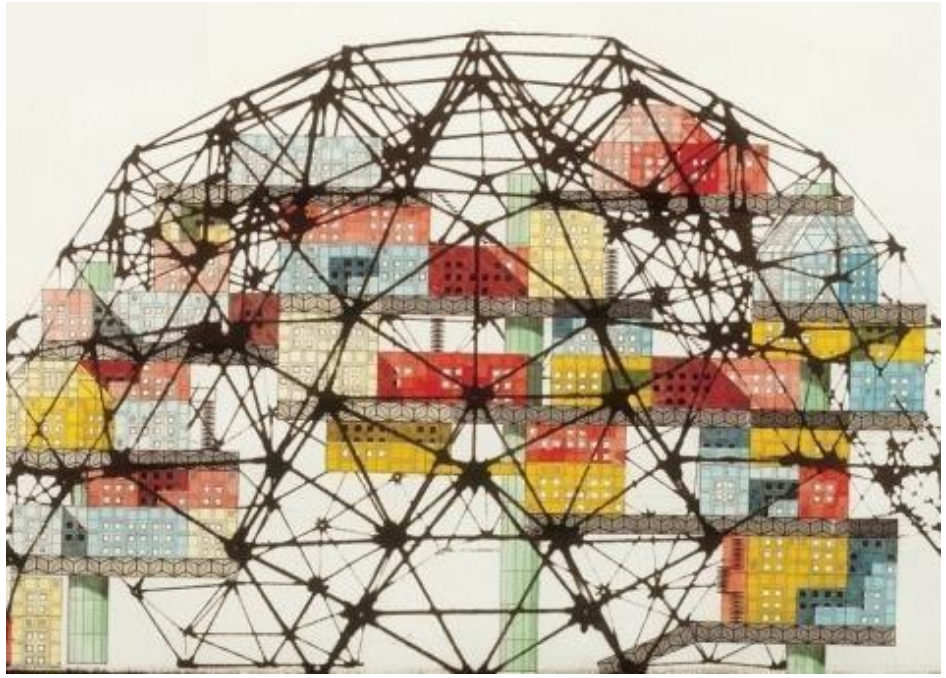


# Event Storming

Pratique très intéressante pour identifier les contextes bornées



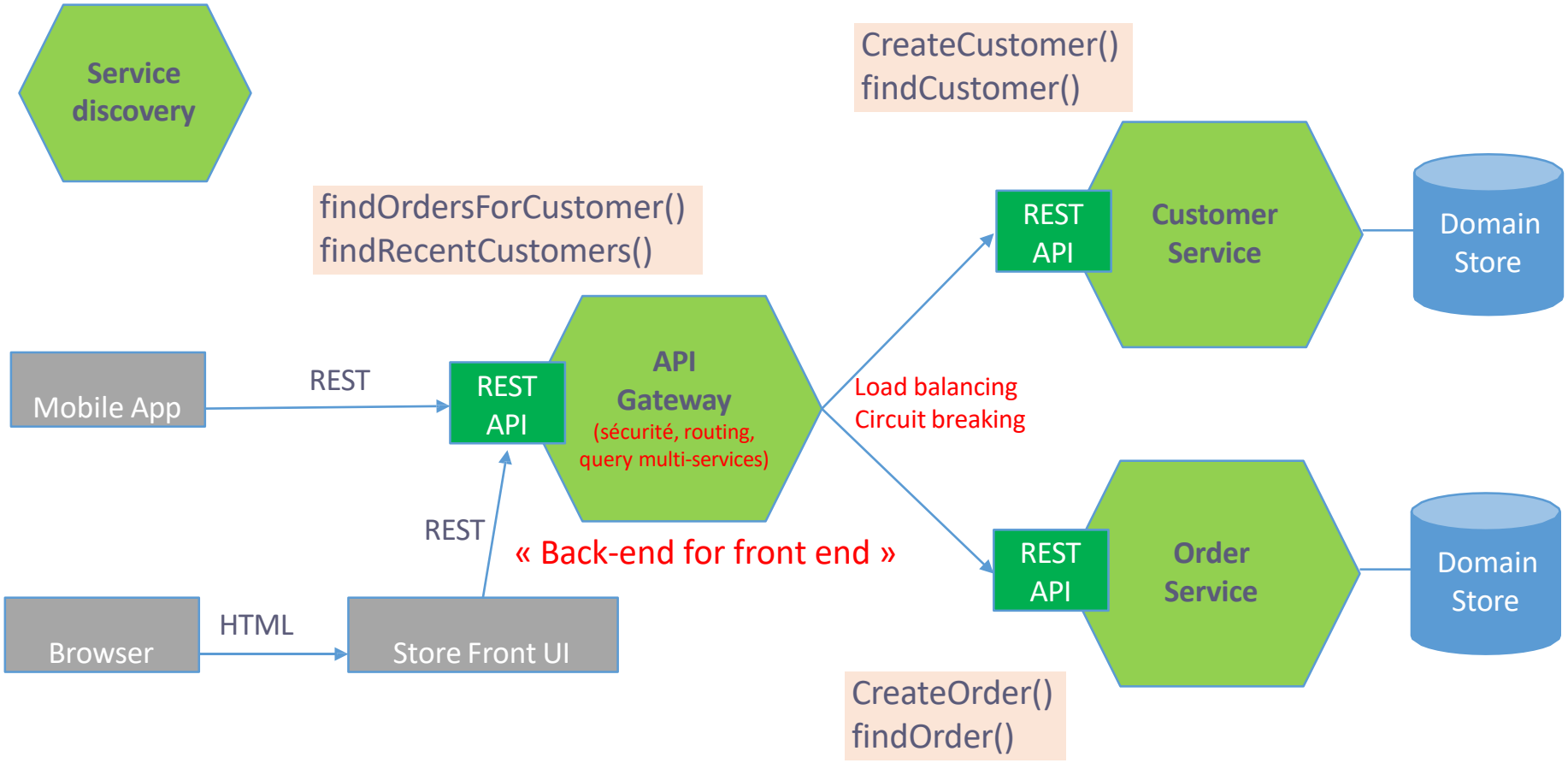
# Différents patterns d'architecture



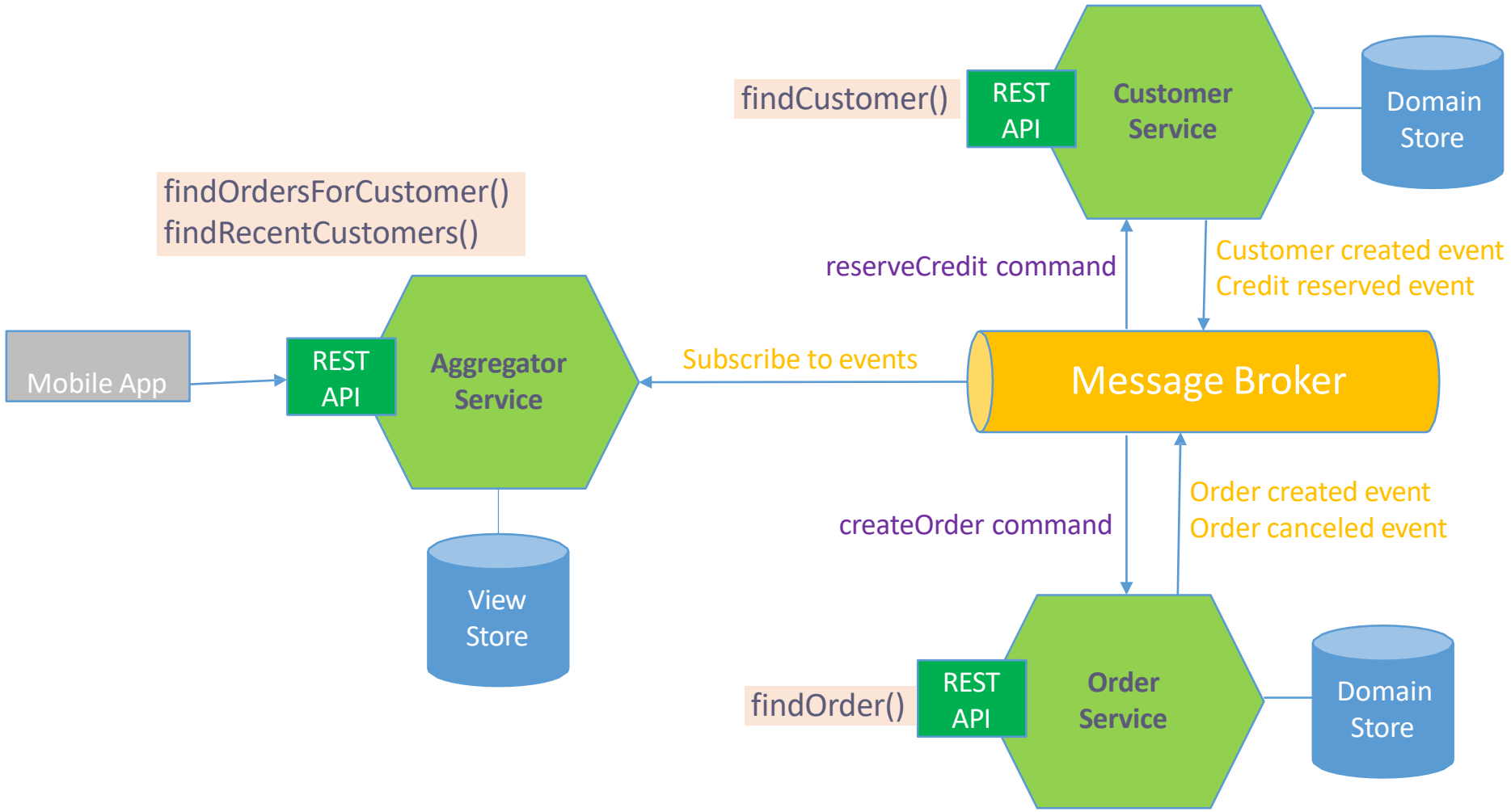
Pattern = Solution réutilisable répondant à une problématique

Exemples :
Comment gérer la communication entre les MS ?
Comment accéder aux données distribuées ?
Comment assurer la robustesse ?

# 1 DB by Service / public API / synchronous communications



# 1 DB by service / Async messaging / commands & events



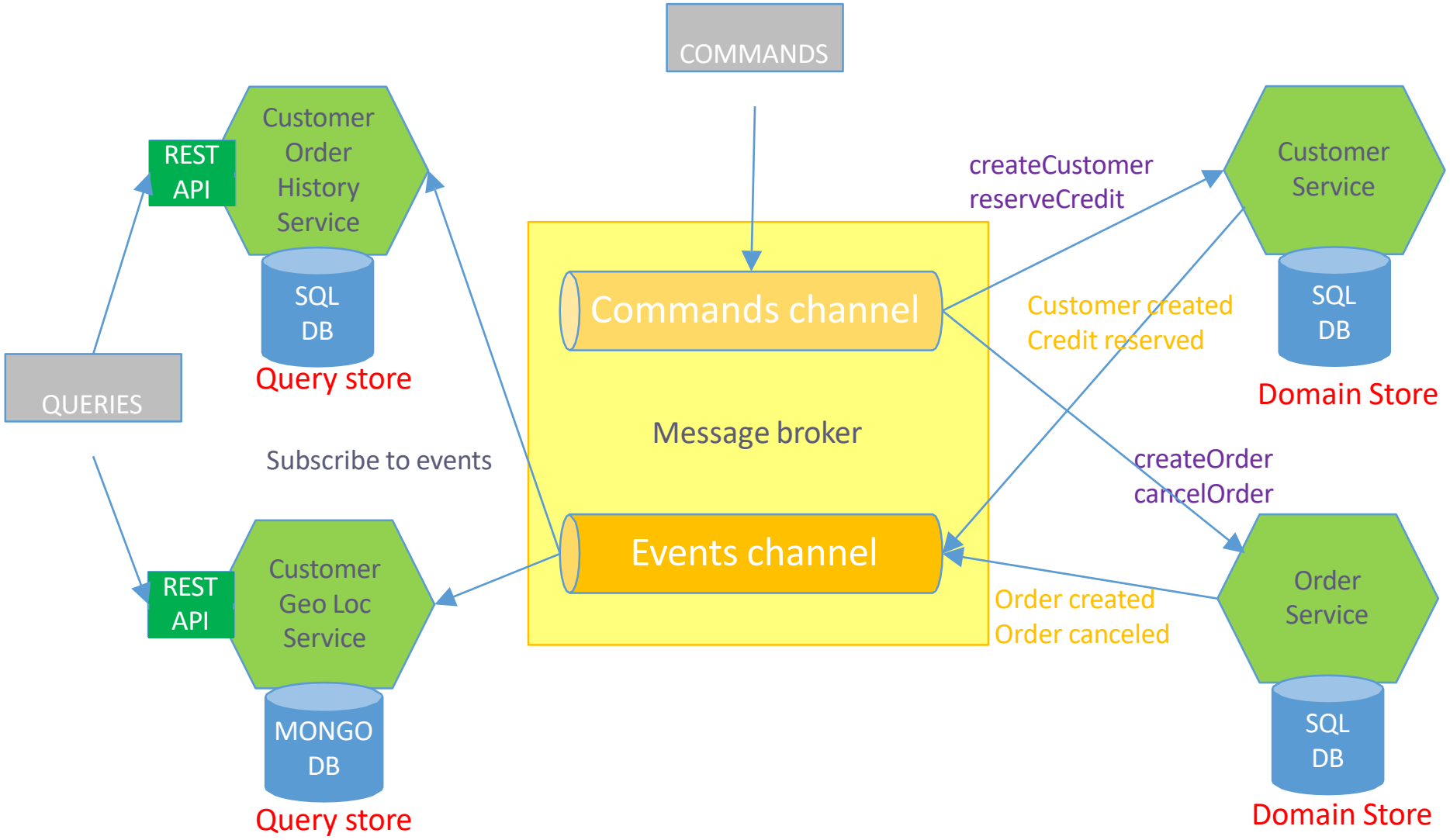
# CQRS pattern: Command Query Responsability Segregation

- Séparation des composants d'écriture (« command») et de lecture (« query »)
- Solution alternative au CRUD
- Write: Opérations sur un modèle métier (DDD)
- Query: Vision aggrégée des objets mis à jour via des events
- Query: Eventual consistency

**Permet:**

- Solution pour requêter des données cross-services
- Meilleure isolation des responsabilités
- Différentes technologies
- Writing DB != Reading DB

# CQRS pattern: Schéma



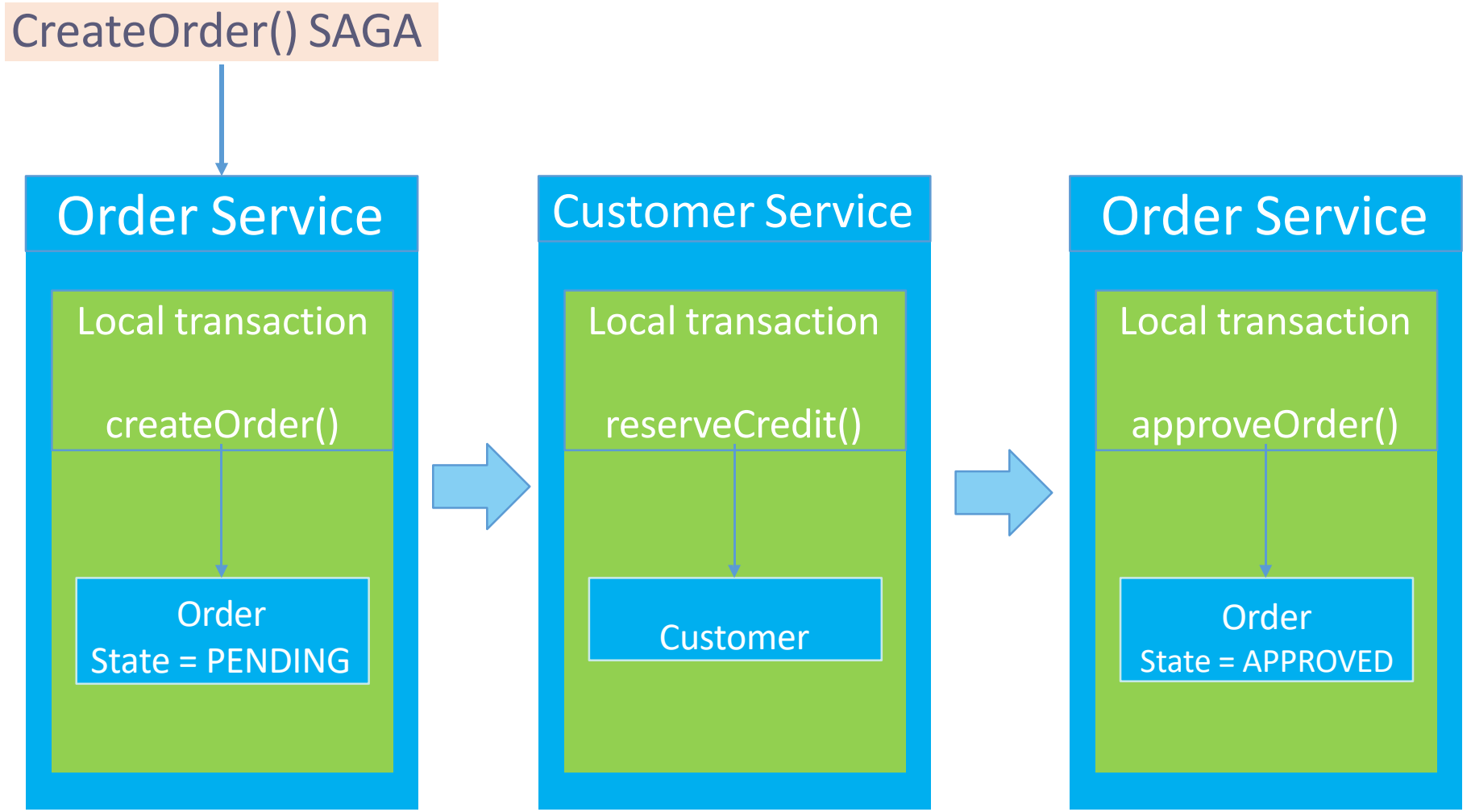
# Collaboration entre microservices

- Comment peuvent collaborer les microservices ?
- Comment implémenter une transaction distribuée ?
- Comment maintenir la cohérence des données distribuées ?

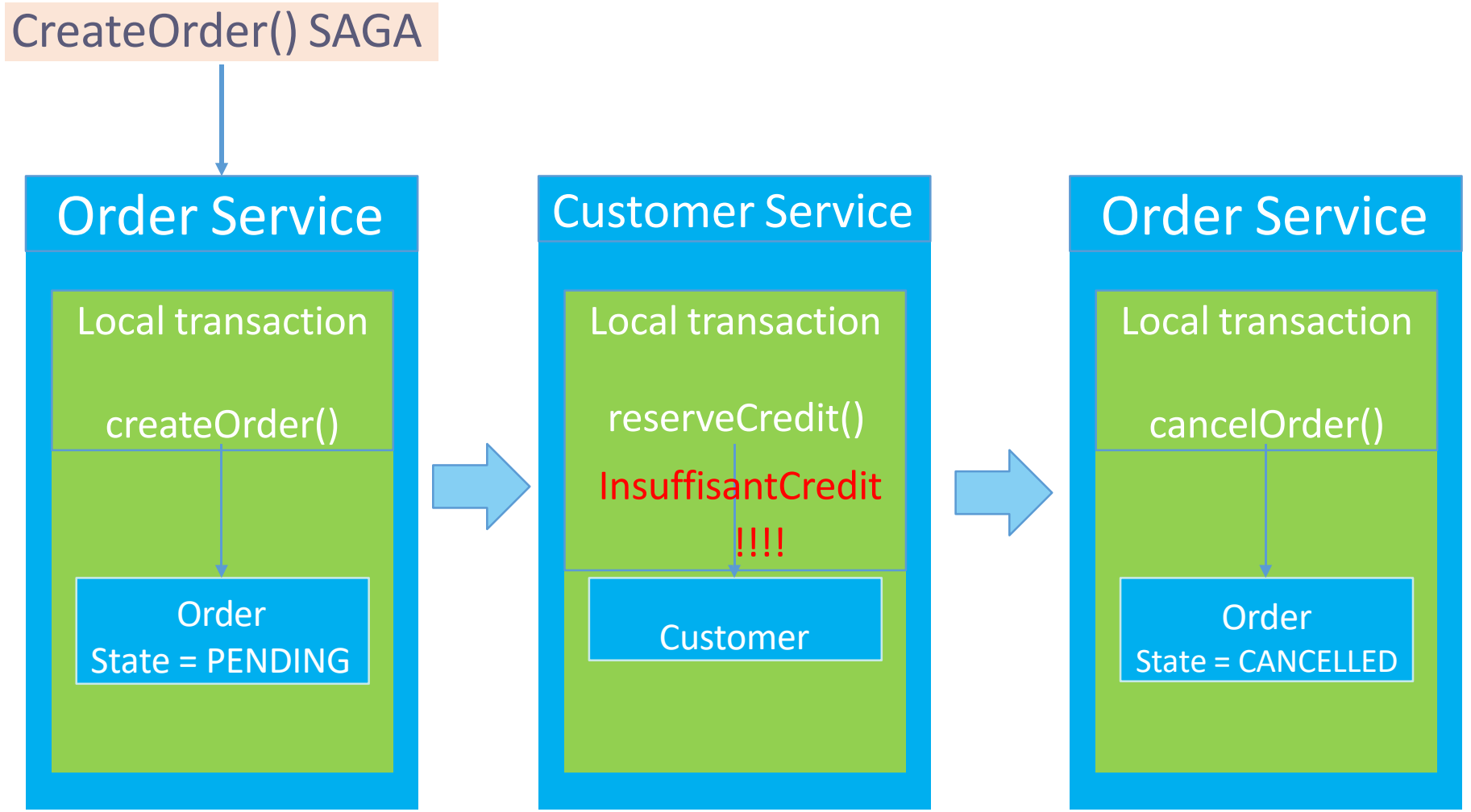
Transaction distribuée 2PC à proscrire

Solution = SAGA PATTERN

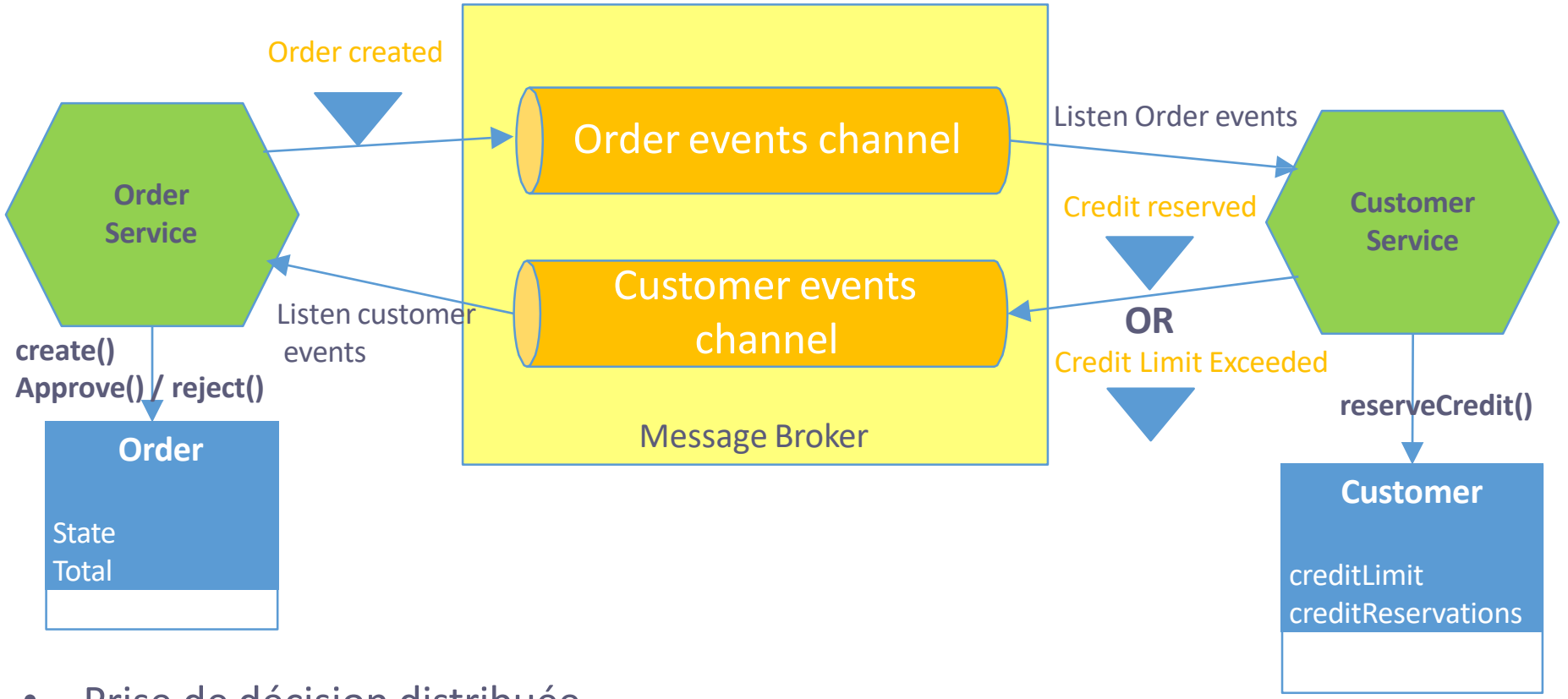
# SAGA pattern: Séquence de transactions locales



# SAGA: Transactions compensatrices

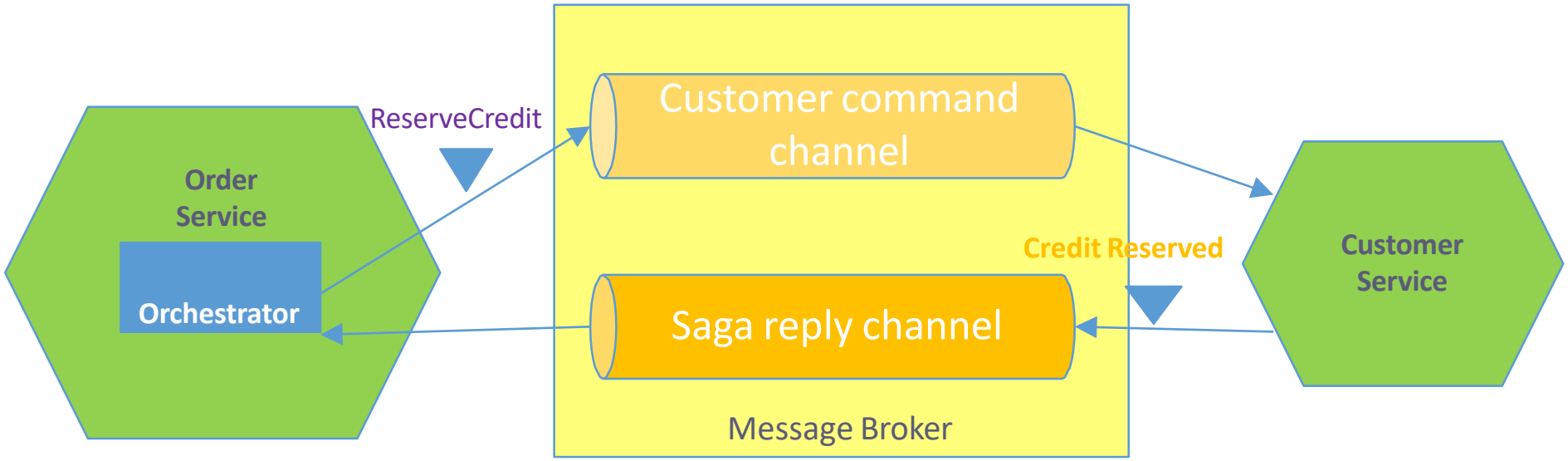


# SAGA: Events Choregraphy pattern




- Prise de décision distribuée
- Coordination basée sur des messages « events »
- Chaque service va écouter les évènements et faire avancer le workflow

# SAGA: Command / Reply Orchestration pattern




- Prise de décision centralisée par un Orchestrateur
- Coordination basée sur des messages « command » et « reply »
- Objet persistant qui traque l'état de la SAGA et invoque les MS

Comment assurer l’atomicité du changement d’état en base et l’envoi de l’event associé ?

# Event sourcing pattern

- Approche « Event driven »
- Persister les objets métiers comme une séquence d'évènements
- Objet métier = State change event
- Rejouer les events pour re-recréer une image de l'objet
- Event Store: Data Store + Message broker
- Se marie bien avec CQRS

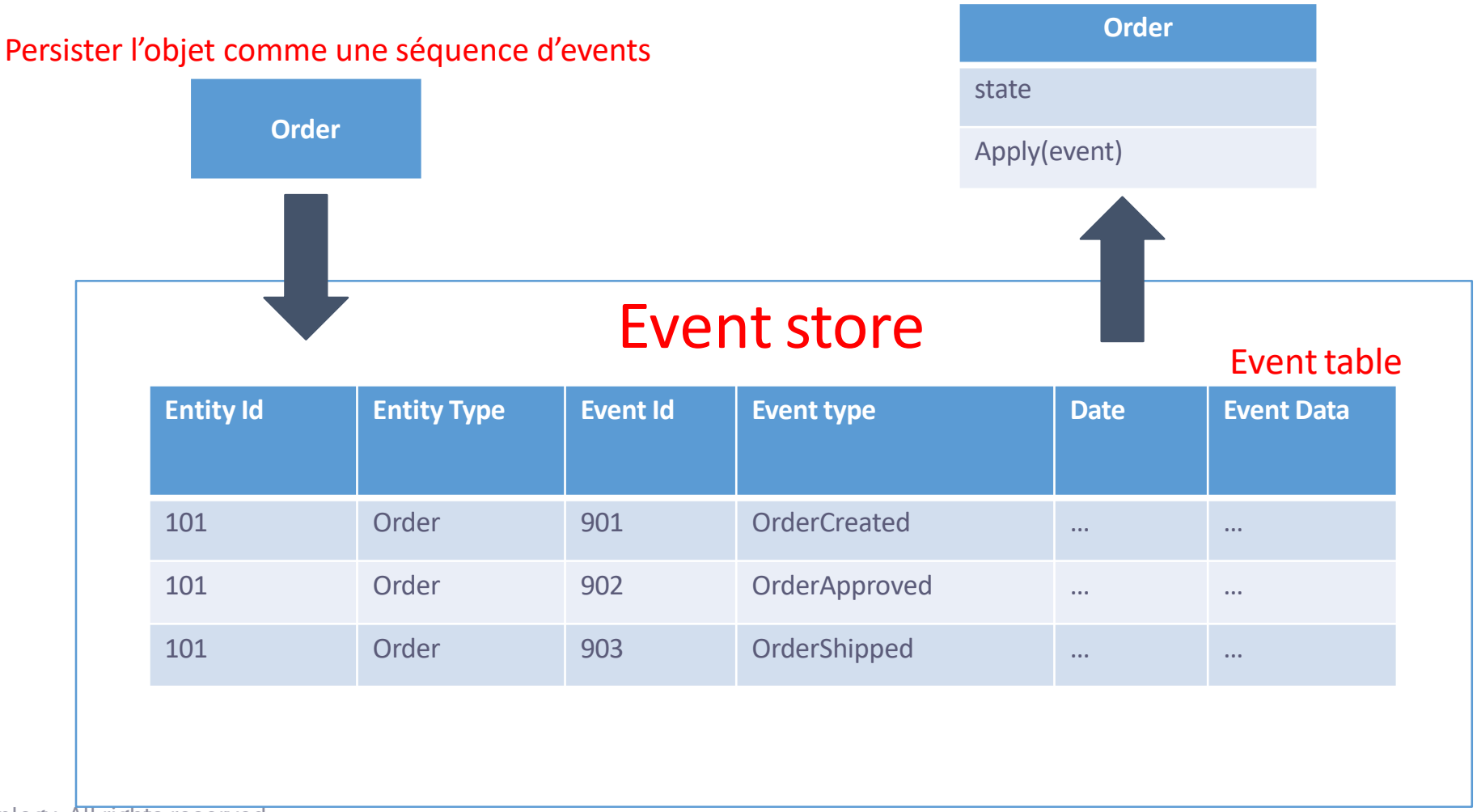
**Permet:**

- Changement d'état en base et envoi de l'event atomique
- Historisation complète
- Audit log + Rejouer les events

# Event sourcing pattern: Schéma

Rejouer les events pour obtenir une image de l'objet

Persister l'objet comme une séquence d'events




| Entity Id | Entity Type | Event Id | Event type | Date | Event Data |
|---|---|---|---|---|---|
| 101 | Order | 901 | OrderCreated | … | … |
| 101 | Order | 902 | OrderApproved | … | … |
| 101 | Order | 903 | OrderShipped | … | … |

# Comment déployer les microservices ?

- Approche Devops:
  - Intégration continue
  - Déploiement continue

- Infrastructure:
  - Déployés dans des containers (ex: Docker)
  - Gestionnaire de containers (ex: Kubernetes)
  - Cloud privé (ex: Openshift) vs Cloud Public (GCP, AWS, Azure)
  - Services Mesh (ex: Istio): sécurité, load balancing, routing
- Serverless

# Serverless: C’est quoi ?

- FaaS (function as a service)
- « Custom code that’s run in ephemeral container in the cloud »
- Cloud providers: AWS Lambda, GCP Functions, MS Functions
- Les instances sont crées à la demande et détruire après le traitement
- Dépend fortement de services cloud tiers
- Approche events

# Serverless: Bénéfices et limites

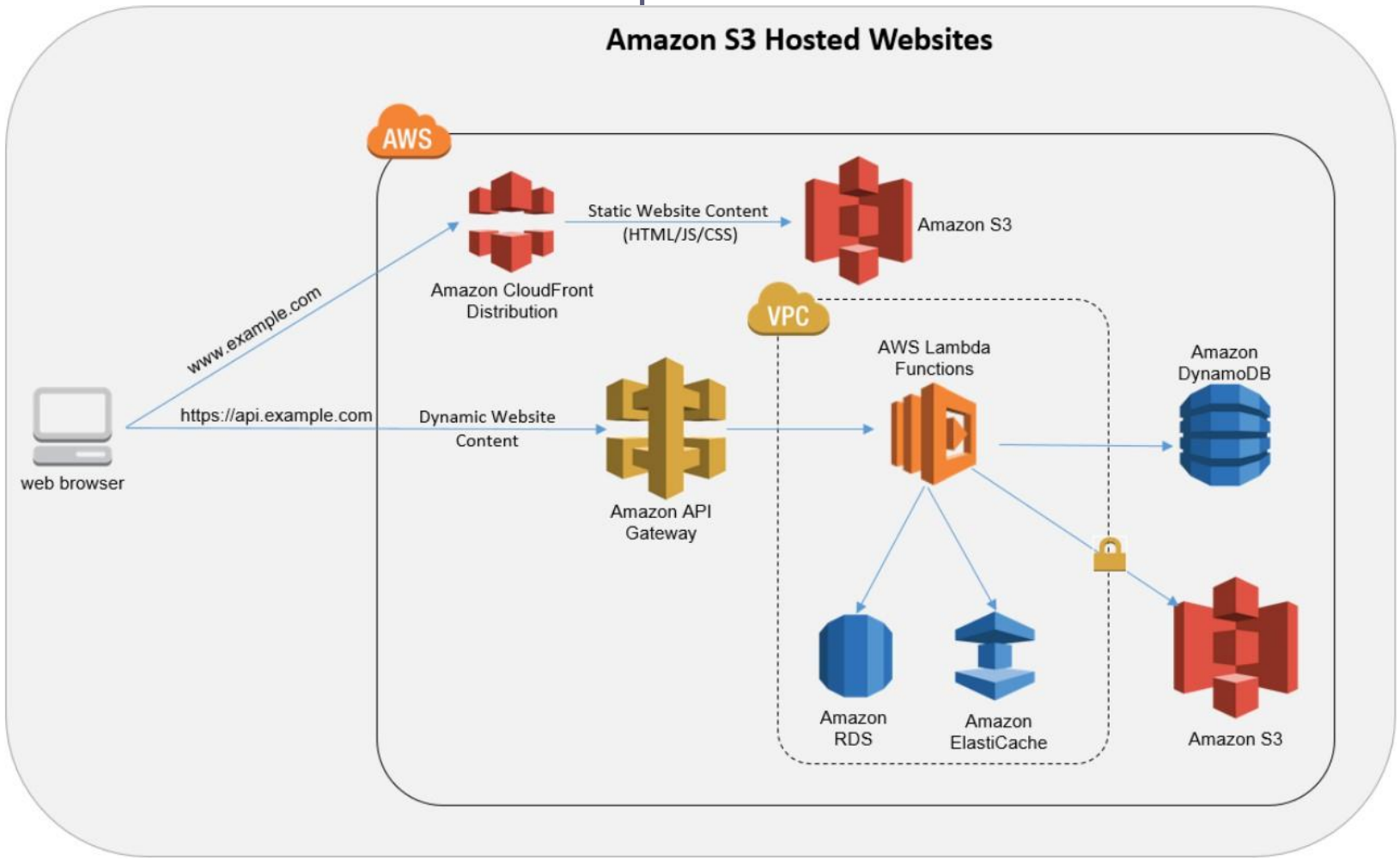
**Bénéfices**

- Auto scaling
- Payer à la demande
- Peu d'infra à gérer
- Event driven
- Sécurité

**Limites**

- Vendor lock-in
- Taille et durée de runtime limité
- Débuggage en local parfois difficile

# Serverless: Exemple



# Cross cutting concerns

| | |
|---|---|
| Load balacing | Ex: Ribbon, Apache, Nginx, HAProxy |
| Logging | Ex: ELK, Logstash, Splunk |
| Circuit breaking | Ex: Hystrix |
| Service discovery | Ex: Consul, Eureka |
| Health check | Ex: Spring Boot Actuator |
| Métriques | Ex: Spring Boot Actuator |
| Configuration | Ex: Spring Cloud Config |
| Proxy / Gateway | Ex: Zuul, Spring Cloud Gateway |
| Tracing | Ex: Spring Cloud Sleuth, Zipkin, OpenTracing |
| Messaging | Ex: Spring Cloud Stream |

Nombreuses Initiatives: Netflix OSS, Spring Cloud, Eclipse MicroProfile

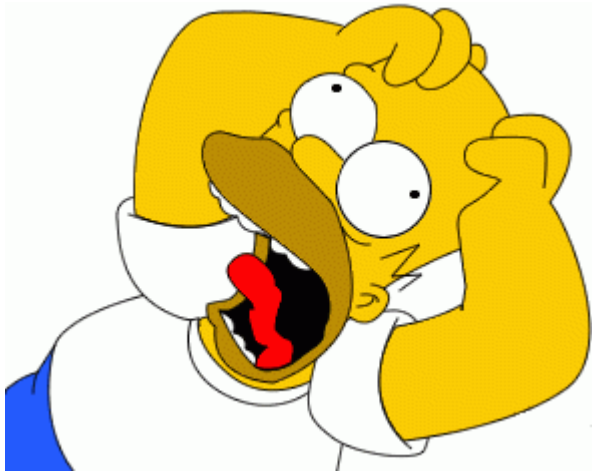
# Bénéfices d’une architecture microservices

- Indépendance – Faiblement couplé
- Travailler en équipes petites, feature teams
- Scalabilité
- Maintenabilité
- Testabilité
- Plus facilement déployable
- Facilite l’innovation et l’expérimentation

# Challenges à relever

## Complexité:

- Processus distribués
- Données distribuées
- Tolérances aux pannes
- Déploiements
- Monitoring
- Nombreux cross cutting concerns
- Nanoservices anti-pattern



## Pré-requis importants

- Approche Devops impérative
- Provisionning automatique

# Références


- https://microservices.io
- https://www.slideshare.net/chris.e.richardson/mucon-not-just-events-developing-asynchronous-microservices
- https://www.slideshare.net/chris.e.richardson/code-freeze-2018-there-is-no-such-thing-as-a-microservice
- https://microservices.io/microservices/news/2018/11/07/microservices-kong-2018.html
- Livre « Domain-driven Design » par Eric Evans
- Livre « Building Microservices » par Sam Newman
- Livre « Enterprise Java Microservices » par Kenn Finningan
- Livre « Microservices pattern » par Chris Richardson
- Hexagonal at Scale, with DDD and microservices! (Cyrille Martraire) https://www.youtube.com/watch?v=xZOO_CksS-E
- Magazine Programmez! – décembre 2018
- https://www.martinfowler.com/bliki/StranglerApplication.html
- https://github.com/kbastani/event-sourcing-microservices-example
- https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/
- https://fr.slideshare.net/nikgraf/introduction-to-serverless
- https://martinfowler.com/bliki/CQRS.html
- https://www.codeproject.com/Articles/339725/Domain-Driven-Design-Clear-Your-Concepts-Before-Yo