

Architecture Logicielle

<https://c4model.com/>

Sommaire

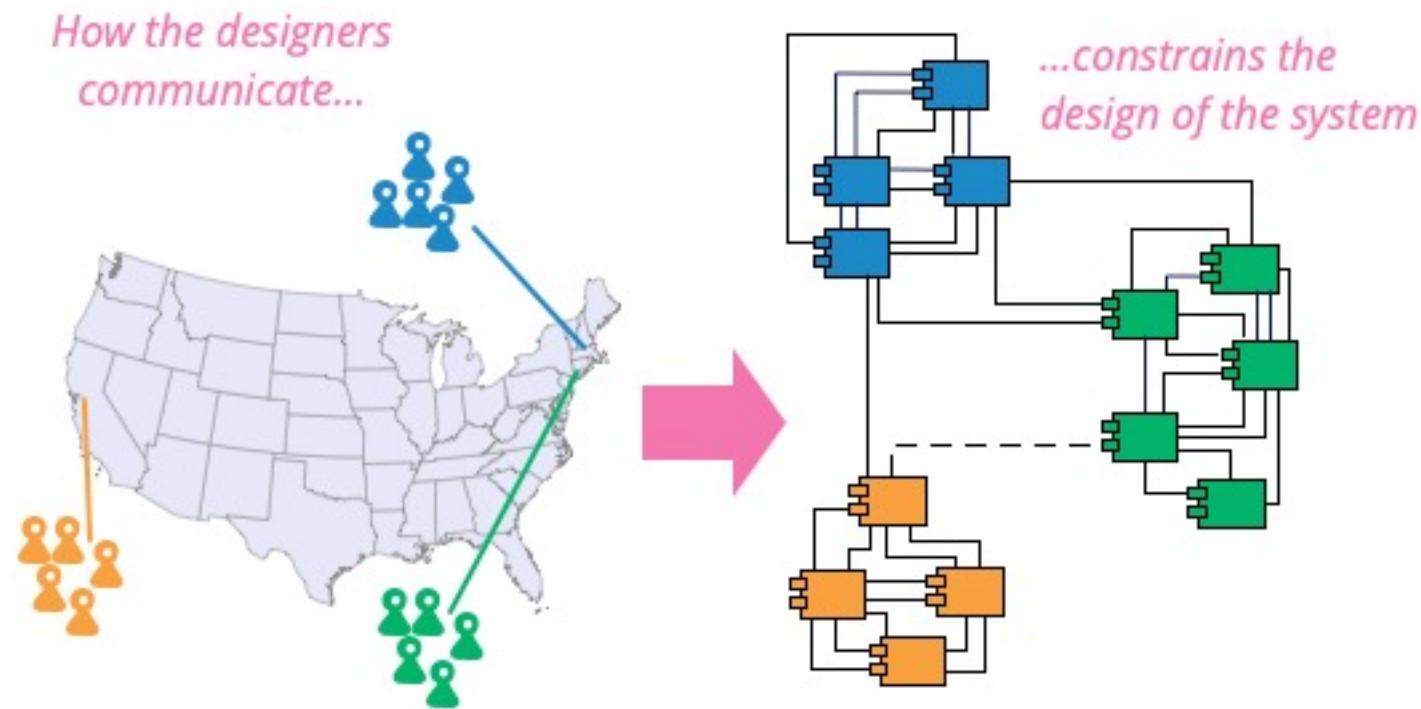
1. Pourquoi ce cours ? Le lien entre Agile et Archi. Logicielle
2. Pourquoi ce cours ? Le lien entre EA et Archi. Logicielle
3. Objectifs de l'architecture logicielle

4. Caractéristiques architecturales
5. Styles architecturaux
 1. Architecture Monolithique et Architecture Distribuée
 2. Présentation de différents styles

1. Pourquoi ce cours

c

Agile et Architecture Logicielle : Conway's Law



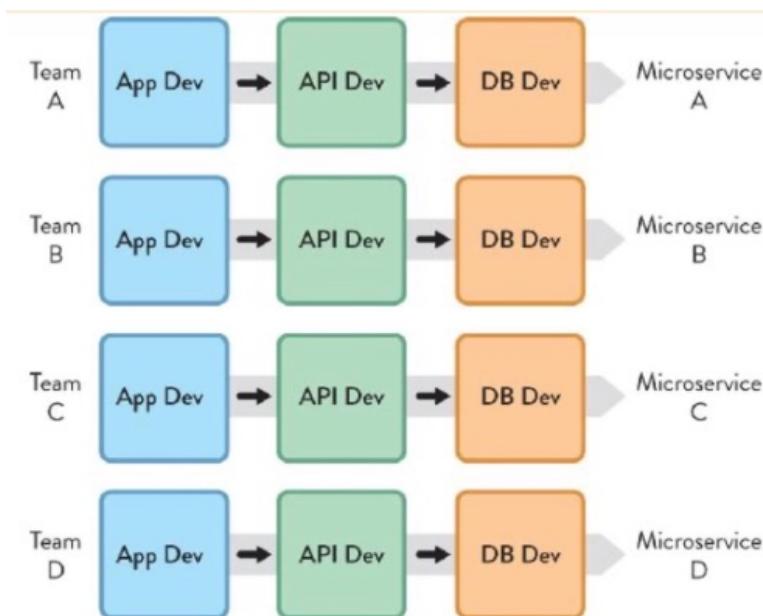
Définition

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

DONC ... Inversion Conway's Law

Inverse Conway's Law

Change the communication patterns of the designers to encourage the desired software architecture.

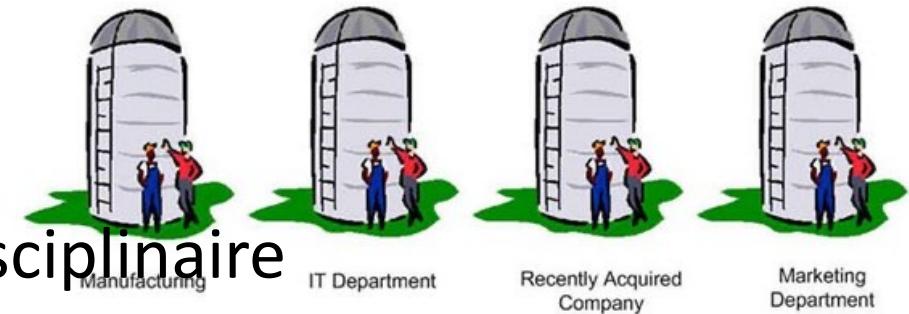


nous pouvons concevoir nos équipes de manière à ce qu'elles « correspondent » à l'architecture logicielle requise

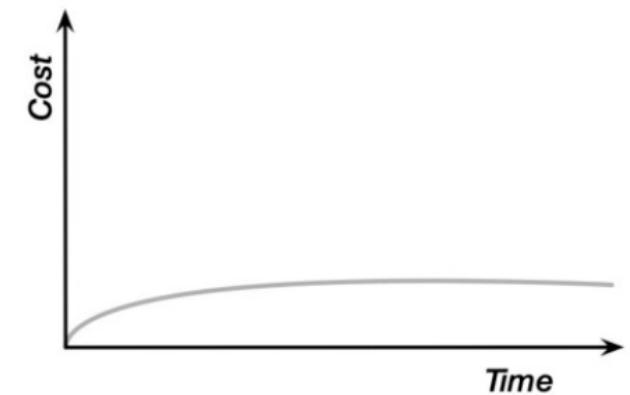
Figure 2.4: Team Design for Microservices Architecture with Independent Services and Data Stores
An organization design that anticipates the homomorphic force behind Conway's law to help produce a software architecture with four independent microservices. (Again, this is basically the diagram in Figure 2.3 rotated ninety degrees.)

Conséquence ... avoir compris Agile

- Casser les silos organisationnels
- Donc créer des équipes autonome et pluridisciplinaire



- Ce qui permet d'itérer rapidement
- Donc d'avoir un coup de développement stable



Objectifs du cours

- Acquérir du vocabulaire
- Comprendre les grands styles architecturaux

=> Pour pouvoir participer à des réunions et proposer des axes d'amélioration de l'organisation.

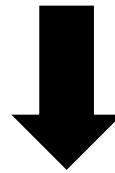
- *Archi dit : « On veut mettre des microservices »*
- *Vous pouvez dire : « Il faut retravailler notre organisation pour casser les silos et créer des équipe autonome cf Inverse Conway Law »*

2. Pourquoi ce cours

Lien entre Enterprise Architecture et Architecture Logicielle

EA Rappels

- Un plan qui lie 4 couches
 - Business
 - Application
 - Data
 - Technologie



Architecture Logicielle
DBMS Architecture
Infrastructure Architecture

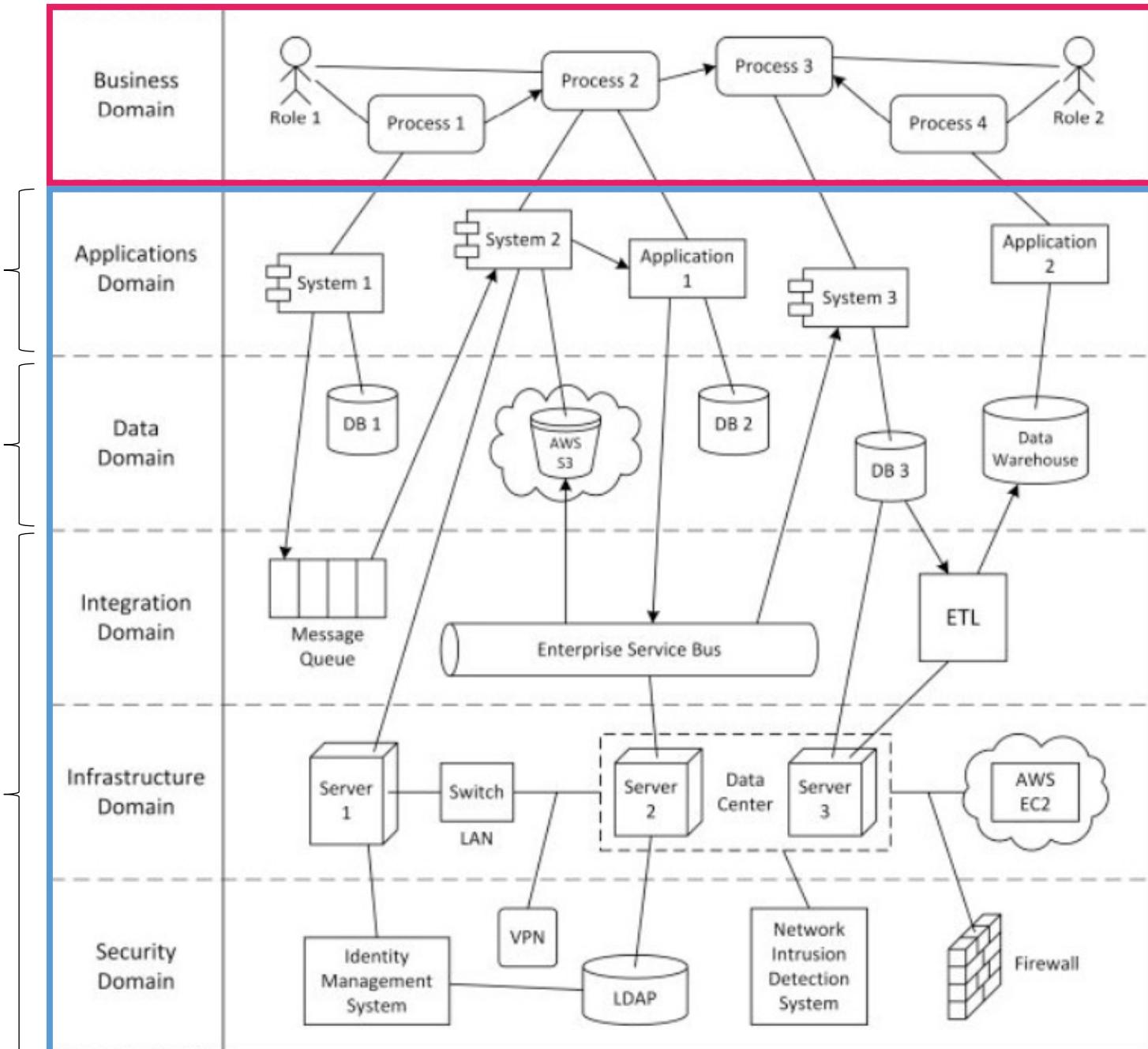
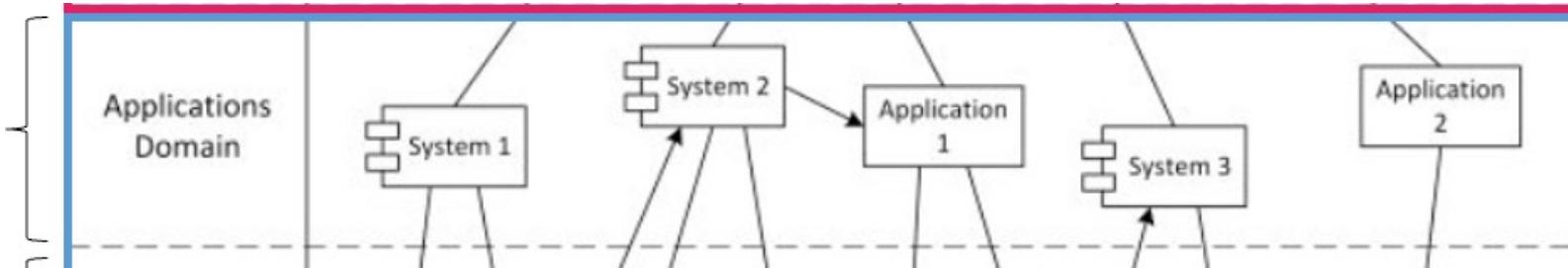
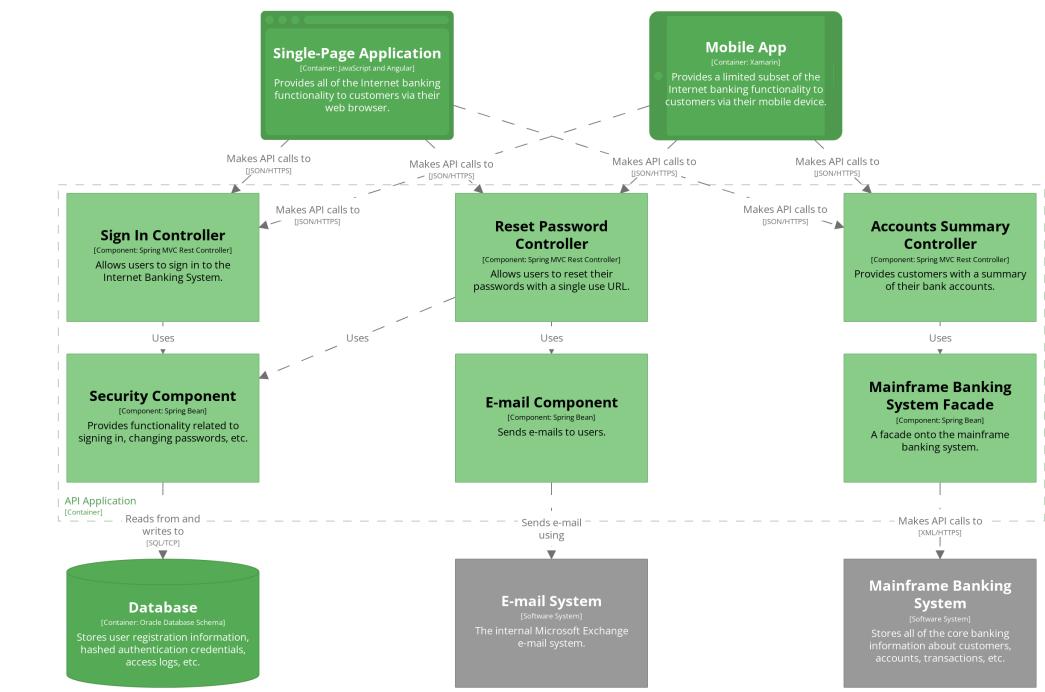
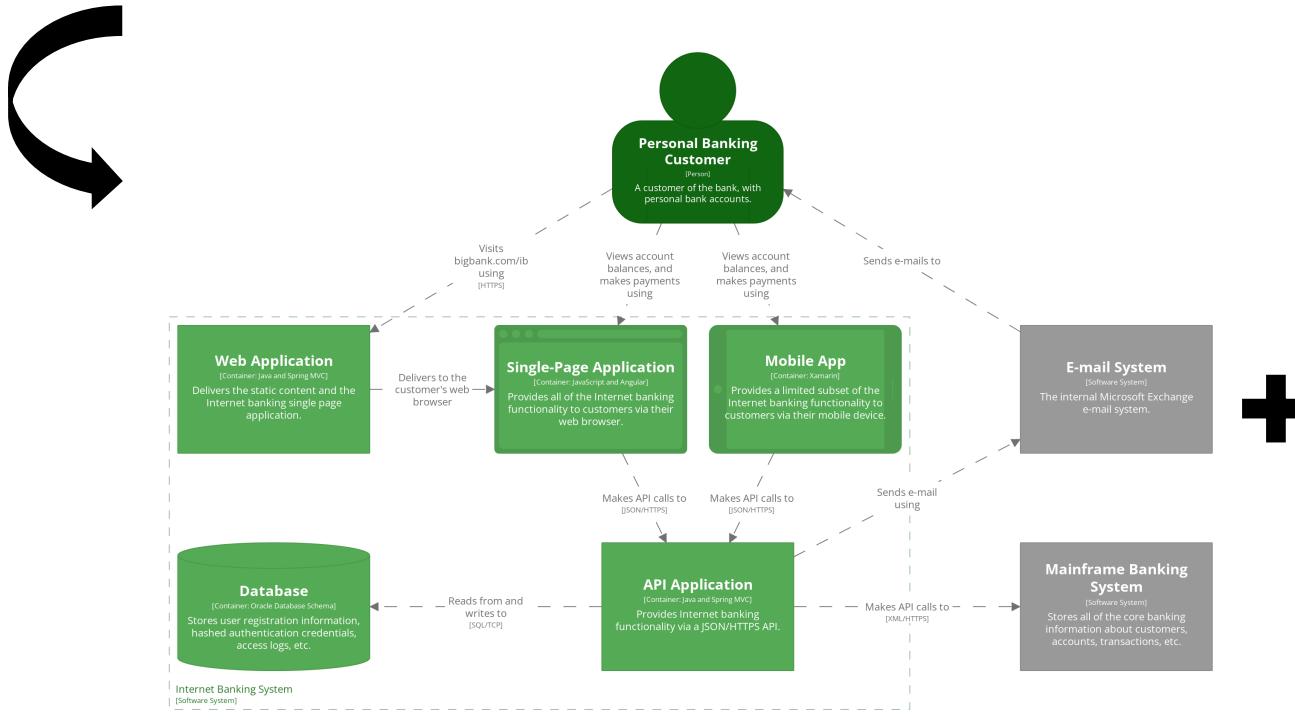


Figure 2.2. EA domains as different layers of an organization

Architecture Logicielle



90% Applications
9% Base de données
1% Infrastructure



3. Objectifs de l'AL

Lier Business et IT

Lier le Business et l'IT => pareil que l'EA

- *E.g. Nous avons un pics d'utilisation de nos service le soir à 20h.*
 - => *Comment construire une architecture logicielle qui répond au métier ?*

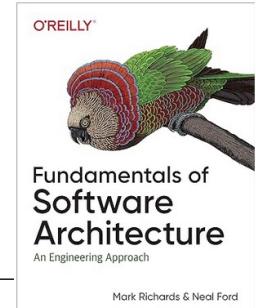
Nous, on va s'intéresser à l'IT



Des concepts intemporels



2003



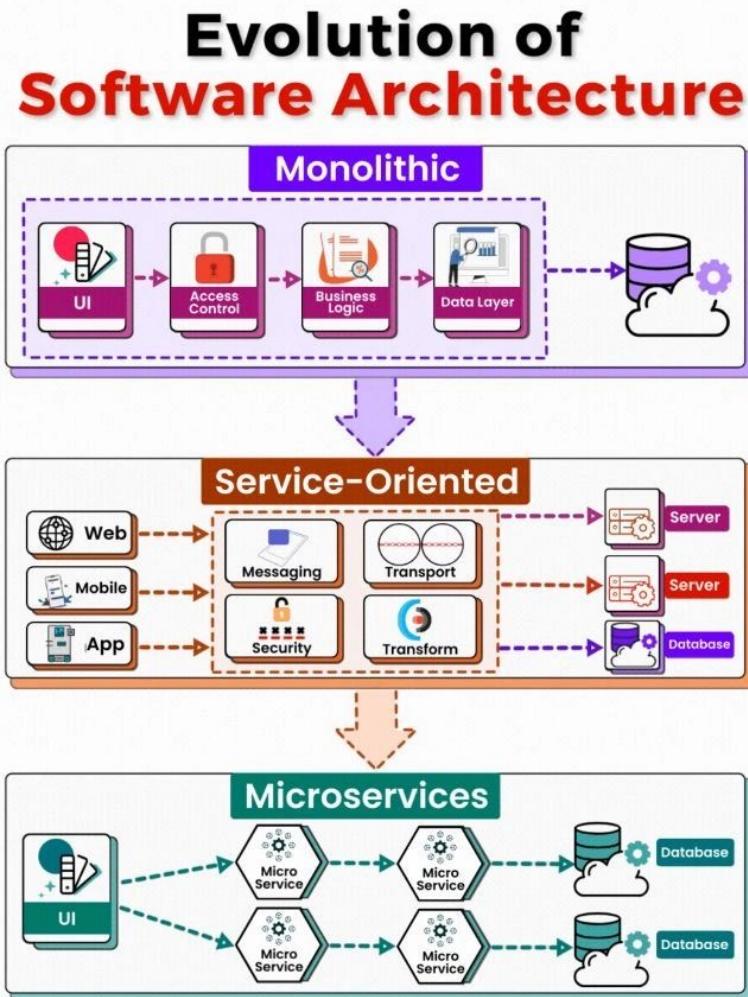
2020

Toujours
d'actualité

2040

L'architecture logicielle du futur nécessite de comprendre celles qui existent, les besoins auxquels elles répondent, les contraintes qu'elles créent

Comprendre les évolutions



- Beaucoup d'évolutions qui entraîne une forme de complexité
- Objectifs : faire des choix éclairé

Faire des choix éclairés

The End of Microservices: Why Companies Are Moving Back to Modular Monoliths

In 2023, Amazon Prime replaced its microservices architecture and moved to a monolith, which helped them cut costs by up to 90% in terms of everything.

Shopify's Implementation of the Modular Monolith: Componentization

4. Caractéristiques architecturales

Objectifs

- Mettre des mots sur les besoins IT (scalabilité, élasticité, etc ...) pour pouvoir choisir une architecture qui répond à ce besoin

Performance

Représente la performance par rapport à la quantité de ressources utilisées dans des conditions données.

Question :

Citez des exemples de « performance »

- Comportement temporel : temps de réponse, débit
- Utilisation des ressources : quantité et types de ressources utilisées
- Capacité : limites maximales

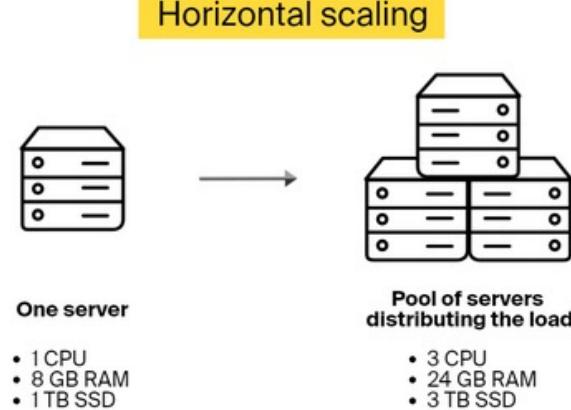
Scalabilité

La *scalabilité* consiste à gérer un grand nombre d'utilisateur sans avoir une dégradation sérieuse des performances.

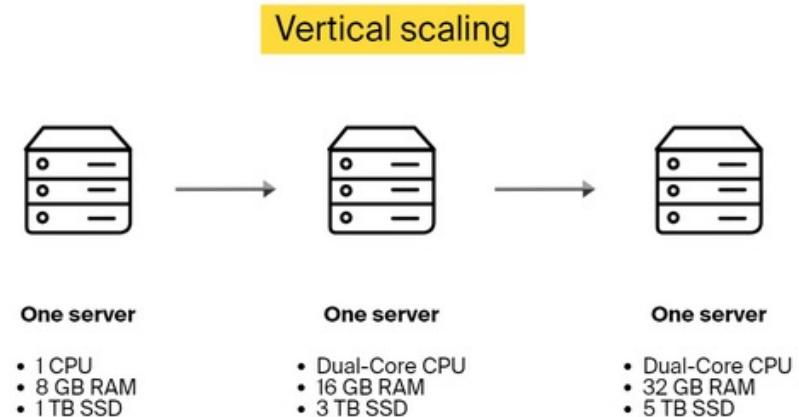
Question :

Comment pourrait-on faire ?

Ajouter nœuds supplémentaires (scale out)



Soit en renforçant le matériel (scale up)



Elasticité

L'élasticité est le degré auquel un système est **capable de s'adapter aux demandes** en approvisionnant et désapprovisionnant des ressources de manière automatique, de telle façon à ce que les ressources fournies soient conformes à la demande du système.

L'élasticité permet de saisir les aspects essentiels de l'adaptation, à savoir :

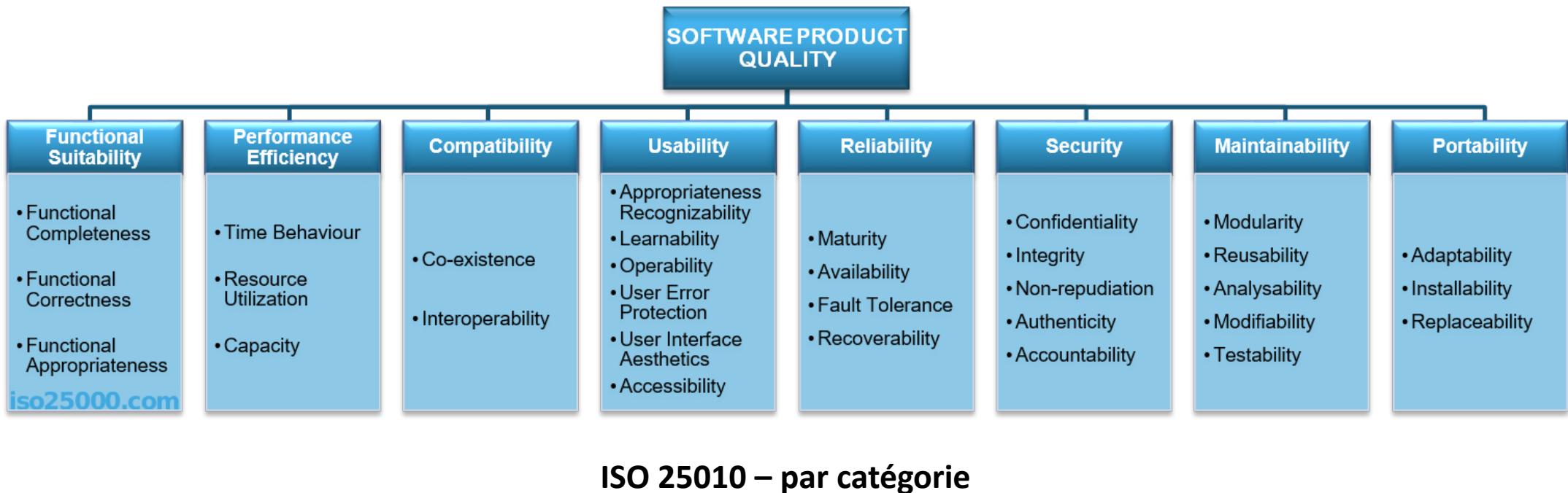
- **La vitesse** : correspond au temps nécessaire pour scale up
- **La précision** : est défini comme l'écart absolu entre la quantité actuelle de ressources allouées et la demande réelle de ressources

Interopérabilité

Est la capacité d'un système d'entreprise (ou de tout système informatique général) à **utiliser** les informations et les fonctionnalités d'un autre système.

- L'interopérabilité est la clé de la construction de systèmes d'entreprise avec des services mixtes.

Beaucoup d'autres



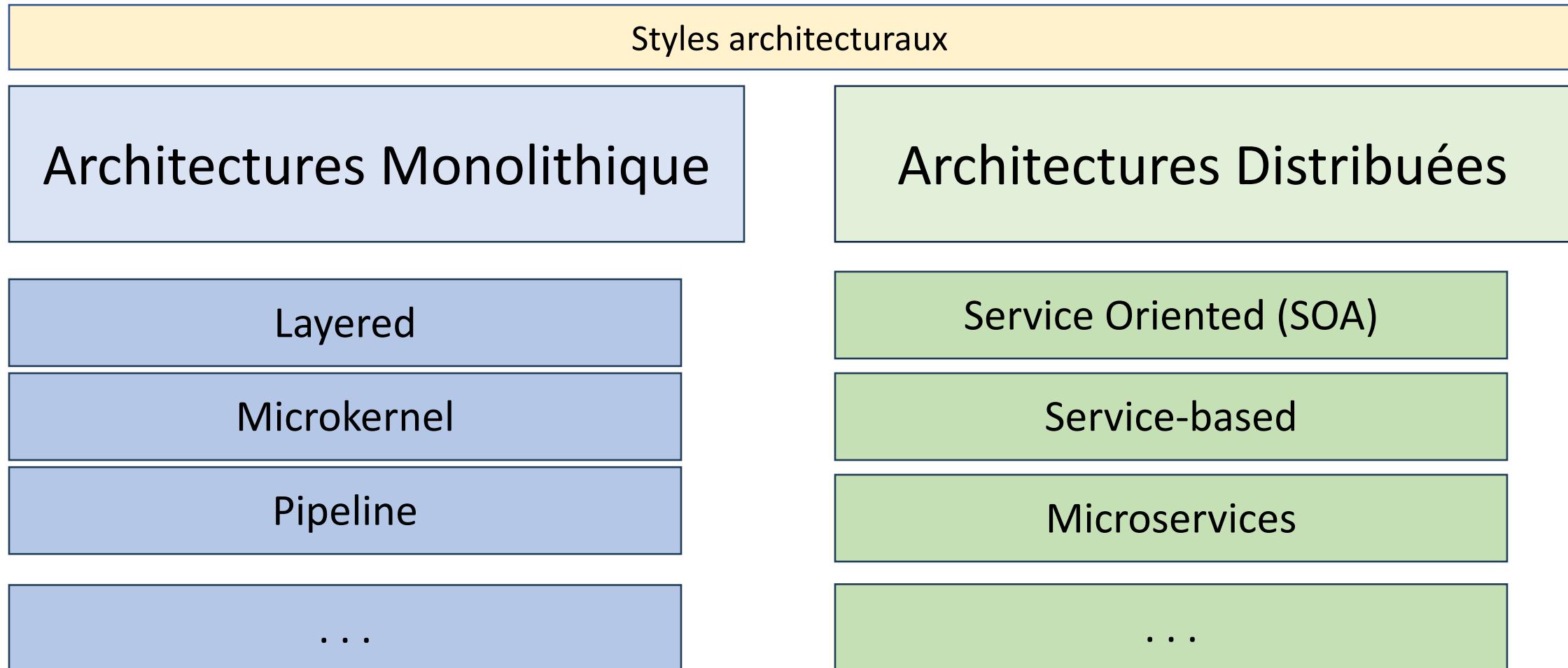
5. Styles architecturaux

Monolithique et Distribués

Objectifs

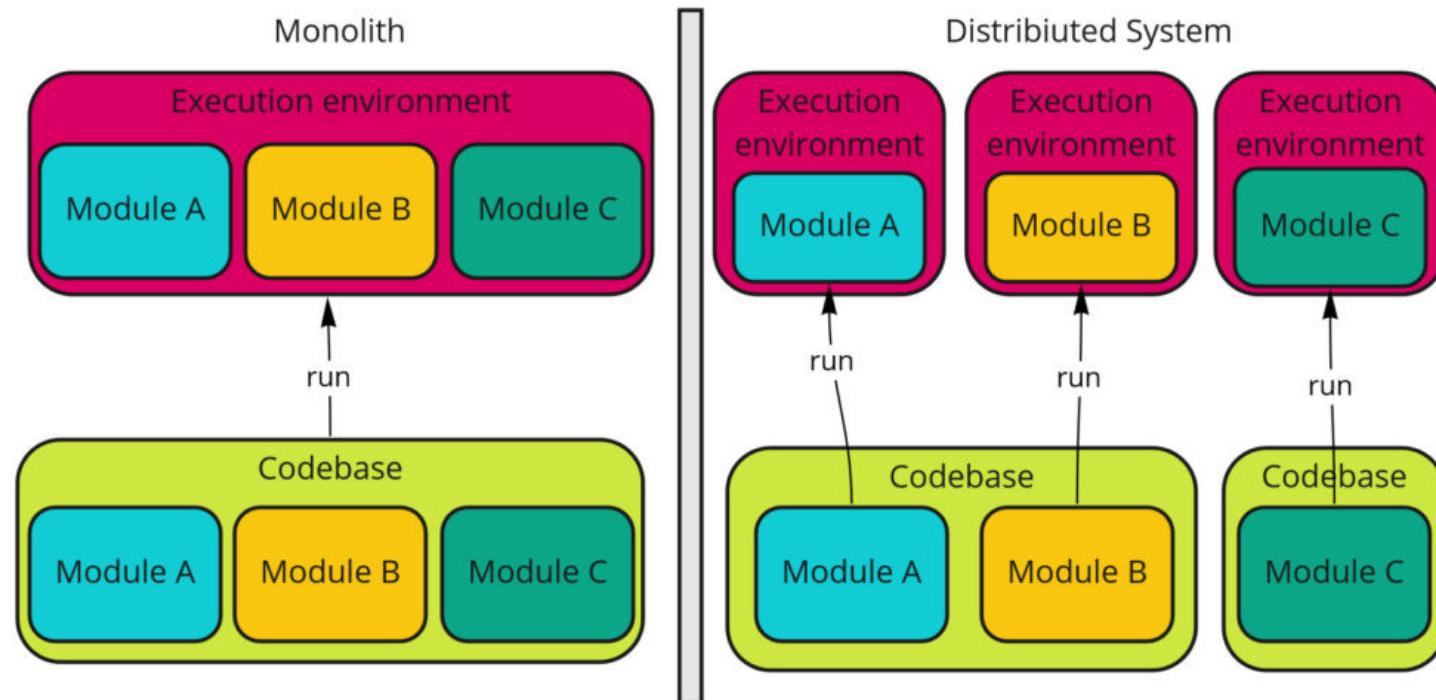
- Découvrir différents styles architecturaux

Deux sous-ensembles



Définitions

- Une architecture monolithique regroupe l'ensemble des fonctionnalités d'une application dans un seul bloc logiciel cohérent (logique métier, bdd). **Une seule unité de déploiement**



- Une architecture distribuée organise une application comme un ensemble de composants ou de services autonomes, déployés sur plusieurs machines ou environnements, qui collaborent via un réseau.

Avantages

- Monolithique
 - Facilité de développement : tout à un seul endroit
 - Facilement déployable : une seule unité
 - Peu couteuse
- Distribuée
 - **Résilience** : si un service tombe les autres non
 - **Scalabilité/Elasticité** : Adapter le nombres de ressource à la charge.

Inconvénients

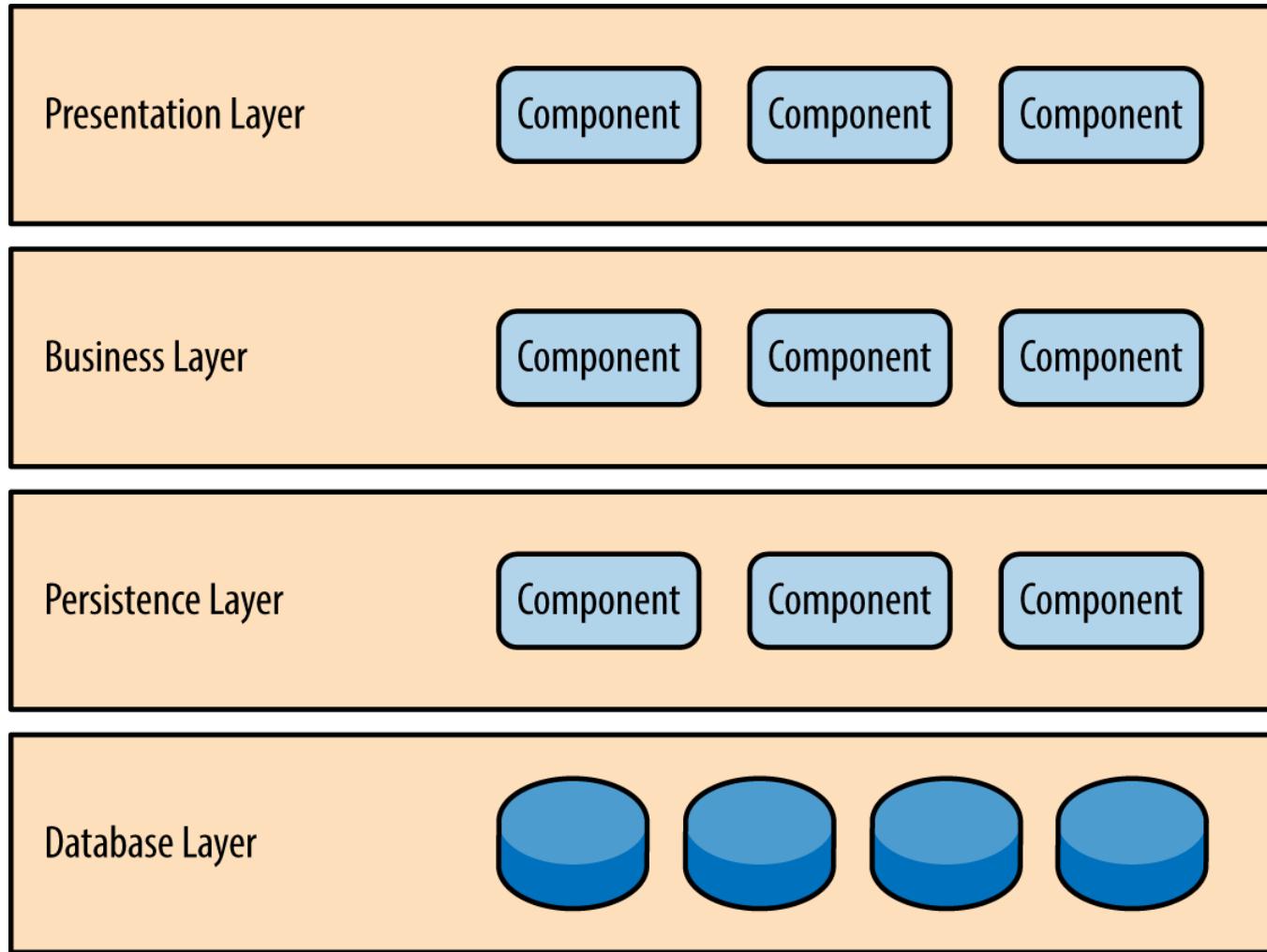
- Monolithique
 - **Tolérance aux pannes** : l'ensemble de l'unité d'application est affectée et tombe en panne.
 - **Scalabilité/Elasticité** : scalar toute l'application
- Distribuée
 - **Complexité** : nécessite bcp de « maturité »
 - **Maintenance élevé** : il faut plus de machines physiques, il faut plus de personnel expert

Architecture Monolithique

#1 Layered
Architecture

Composition

Peuvent être combinées

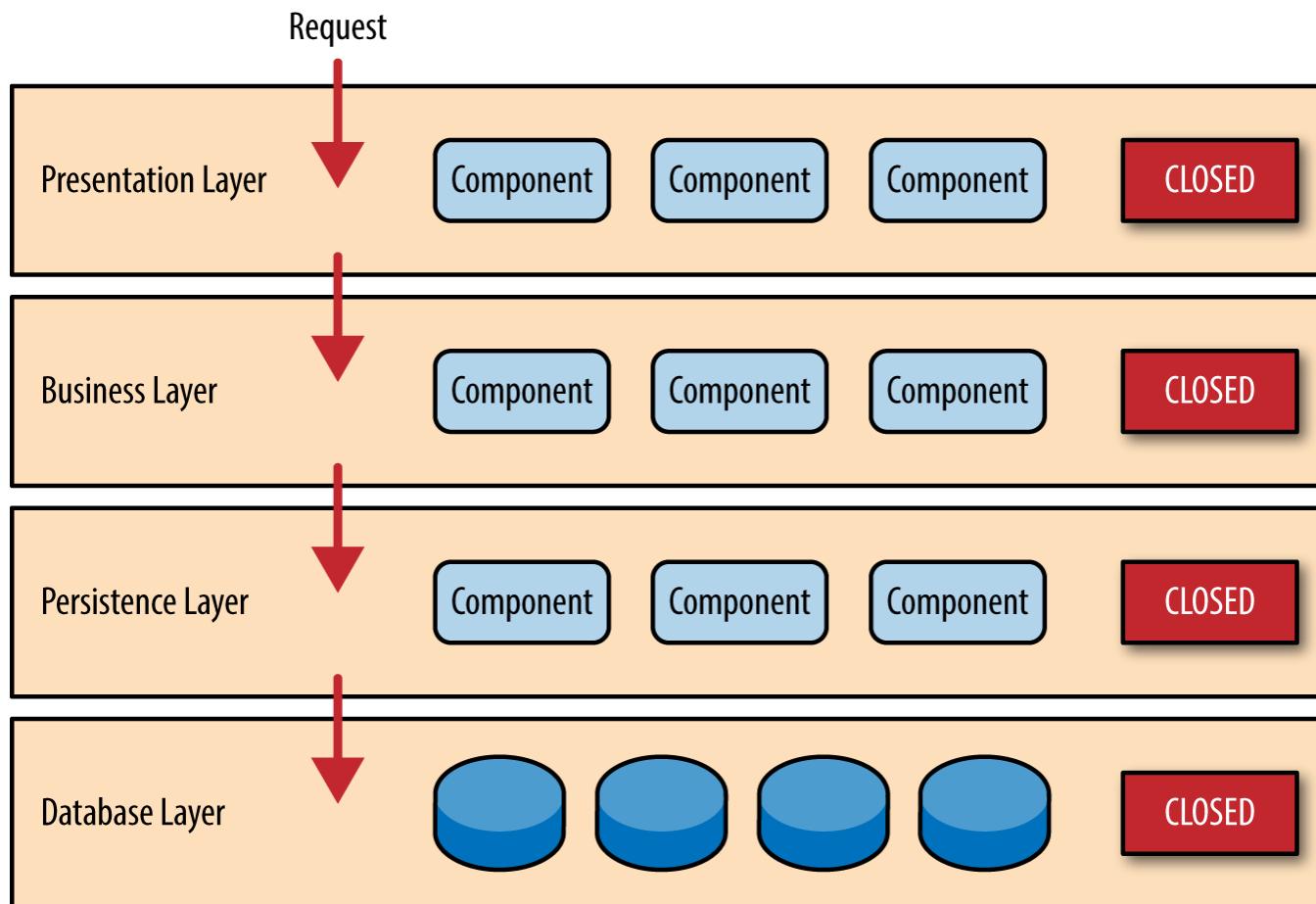


Chaque couche a un rôle bien défini et ne connaît pas les rôles des autres couches

L'IHM ne sait pas comment les utilisateurs sont récupérés de la base de données

separation of concerns

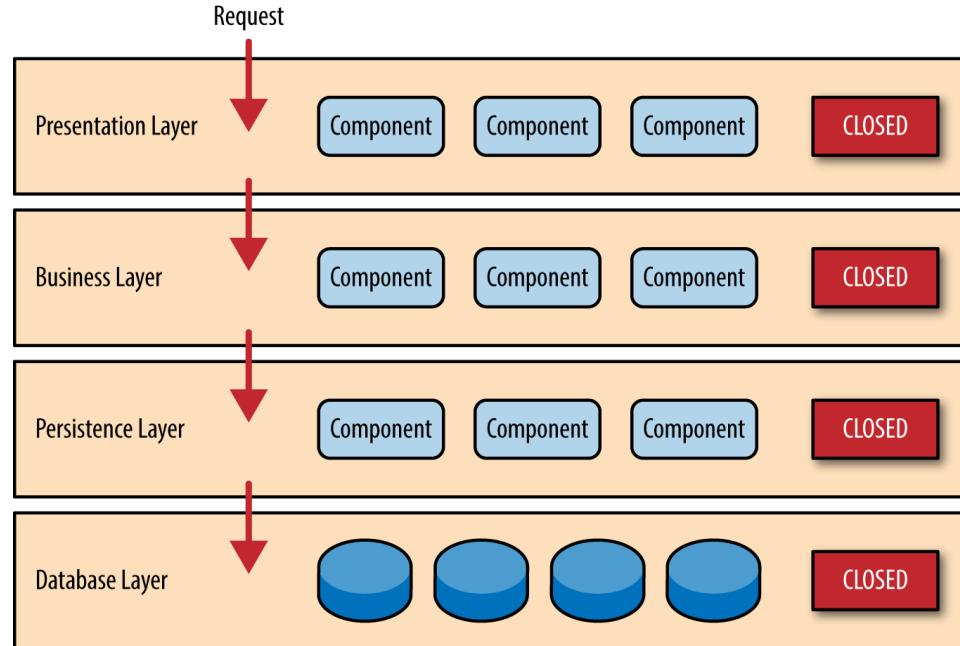
Couche « closed »



Chaque couche est dites **closed**

- Une requête *move from layer to layer*
1. Une requête arrive sur l'IHM
 2. Elle doit passer par
 - Business
 - Persistance
 3. Avant d'interroger la BDD

Pourquoi avoir des couches ?



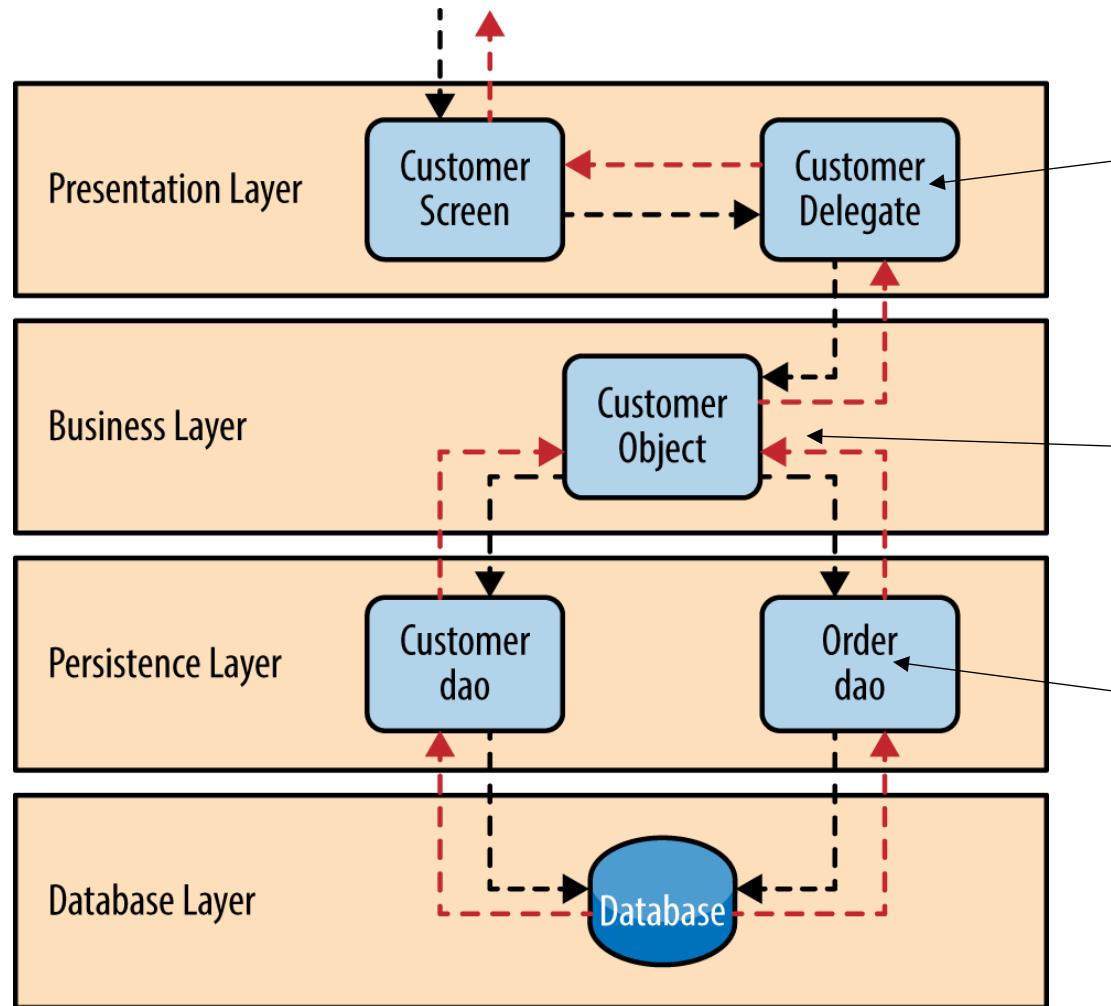
Il est bien plus rapide de passer de l'IHM à la BDD, les couches ralentissent la requête ?

Question :

Pourquoi donc faire des couches ?

- Oui MAIS
 - Concept **Layers of isolation**
-
- Un changement (code) dans une couche
 - N'impactera pas l'autre couche
 - Le changement est **isolé**

Exemple



Knowing which modules in the business layer can process that request (**CONTROLLER**)

Aggregating all of the information needed by the business

Call the DB to get informations (+- SQL request)

*Note : parler DAO et DTO inter-couche
e.g. mdp en double*

Les interfaces !!! D'une couche à l'autre

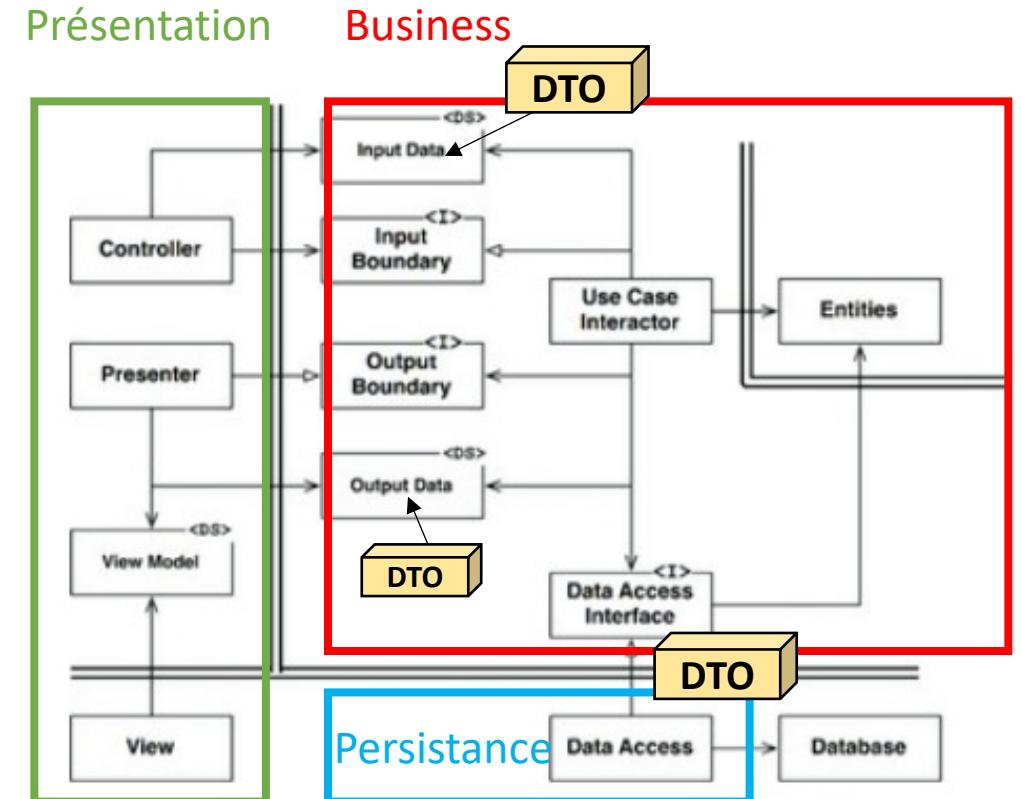
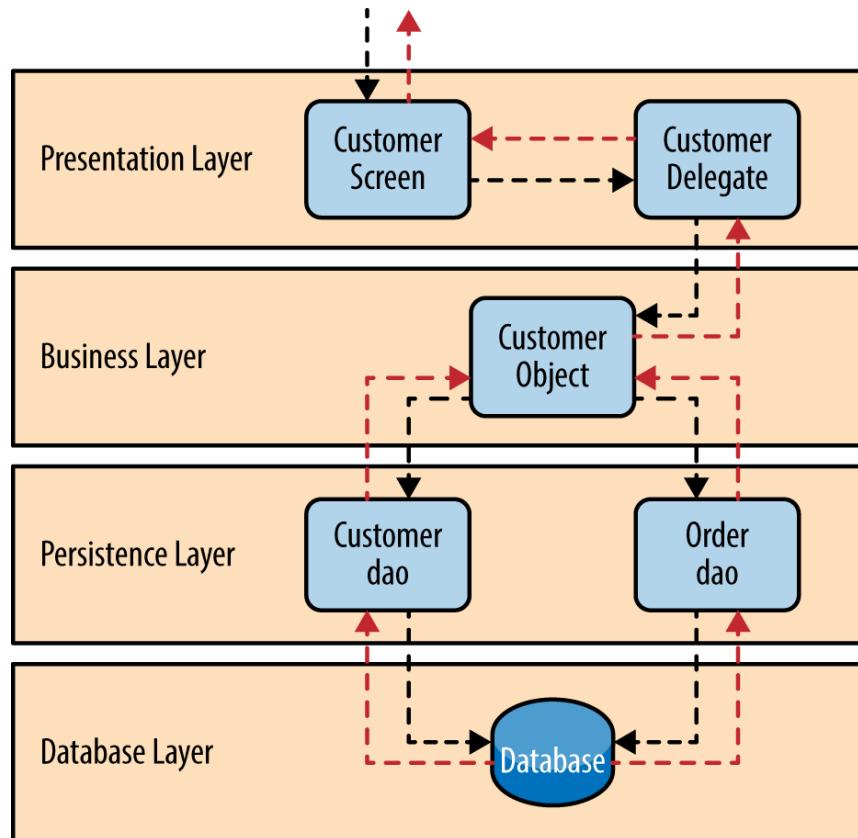


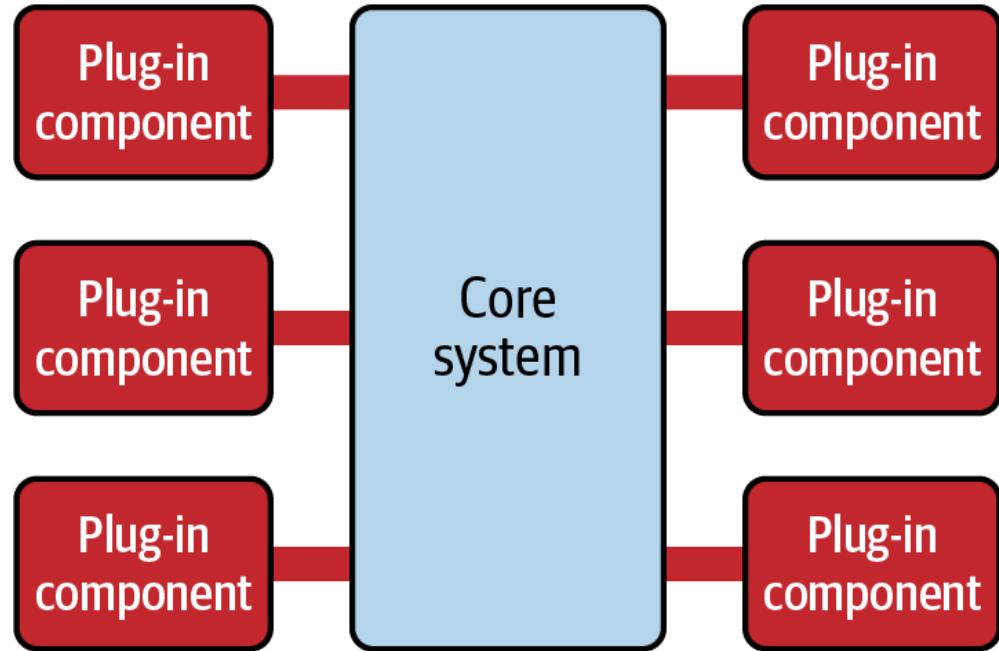
Figure 22.2 A typical scenario for a web-based Java system utilizing a database

La clean Architecture (qui est un patron architectural) fait partie des architecture par couche (comme MVC)

Architecture Monolithique

#2 Microkernel

Composition



Question :

Que pourriez-vous me dire ?

Plugins :

- Devraient être indépendants les uns des autres
- Contient un traitement spécifique
Pour améliorer ou étendre le système

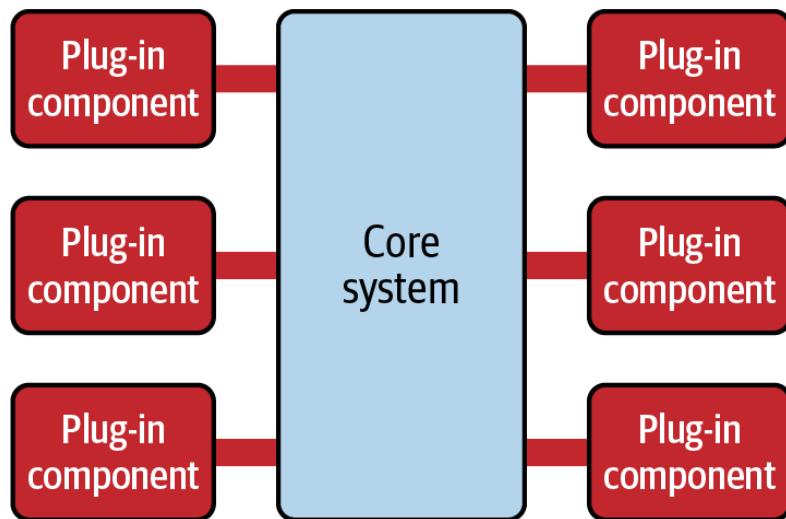
Kernel :

- Cœur applicatif
- Doit connaître l'ensemble des plugins

Communication : la difficulté

La principale difficulté de l'architecture microkernel est la communication:

- entre le cœur applicatif (le kernel) et les plugins
- entre les plugins

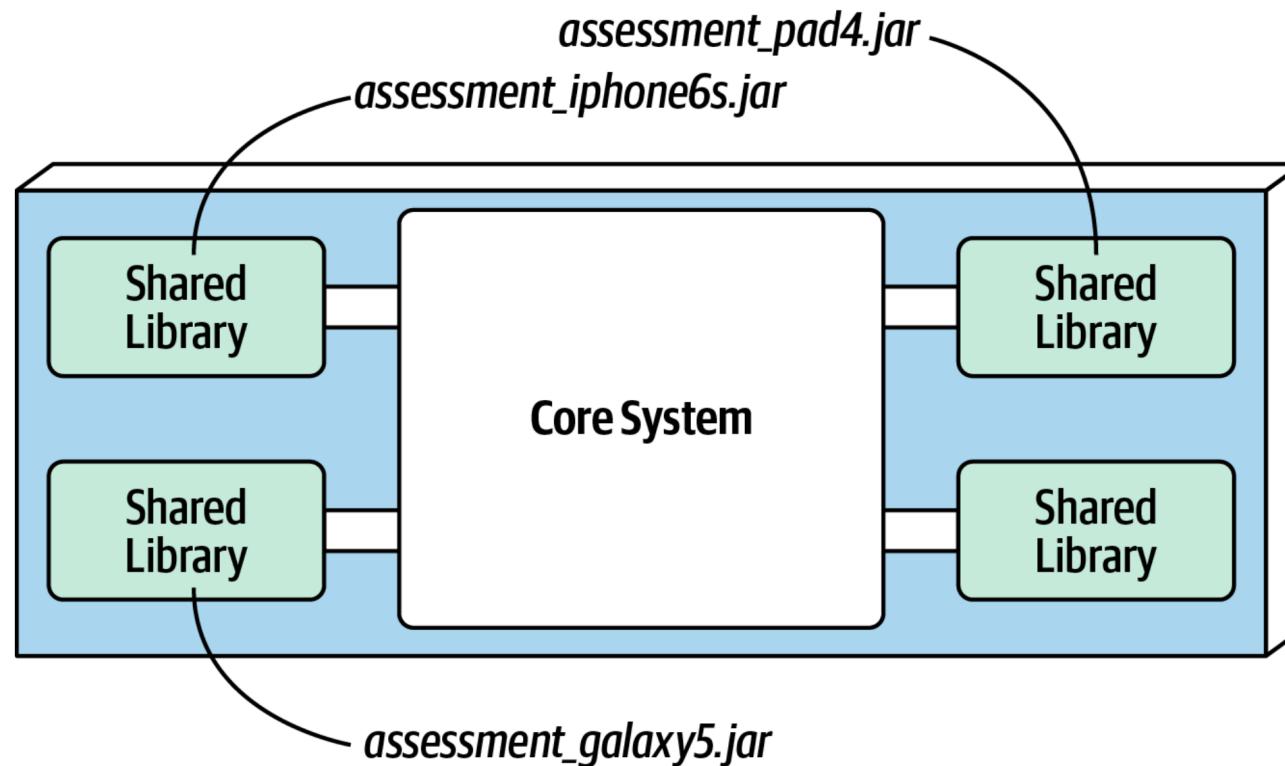


Question :

Quelles solutions pour la communication ?

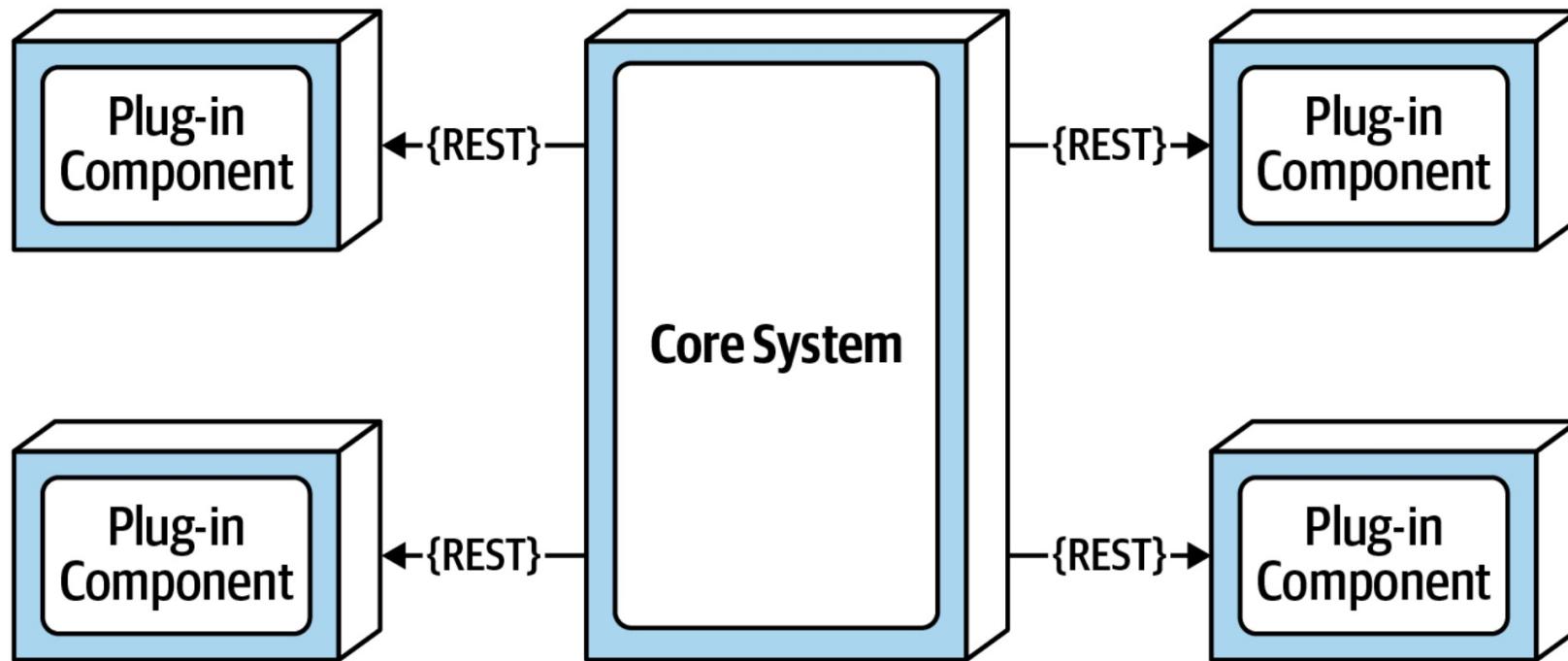
- Point-à-point
- REST
(explications, slides suivantes)

Communication : point-à-point



- le coeur applicatif appelle les plugins grâce un appel de fonction classique via l'interface (i.g. code java)
- le plugin peut appeler le coeur applicatif via l'interface également

Communication : REST

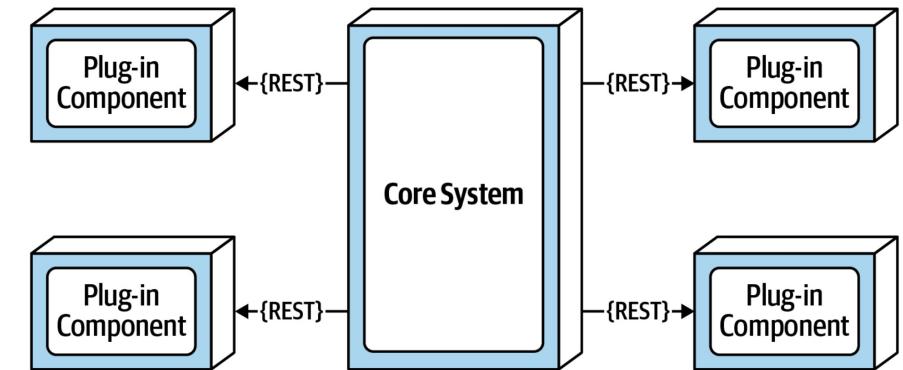


Les modules peuvent également être mis en œuvre comme des services à distance et accessibles par le biais d'interfaces REST à partir du système central.

Communication : REST

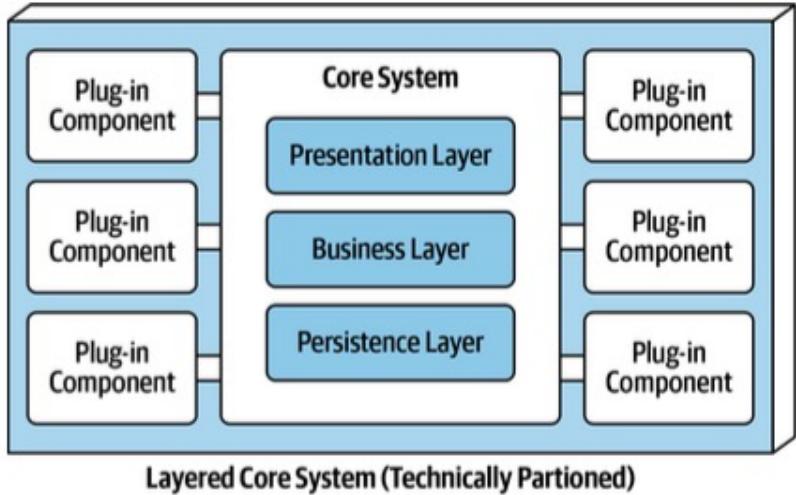
Question :

Que pouvez me dire sur l'architecture microkernel si on opte pour une communication REST ?



- **Indice** : est-ce toujours une approche « que » monolithique ?
- Nous pouvons nous orienter vers une architecture microkernel distribuée
- DONC
 - **la scalabilité** de notre système; si plugin plus utilisé que les autres nous pourrons le scaler
 - d'avoir une **communication asynchrone**; avec l'approche point-à-point communication est forcément synchrone

Implémenter le kernel

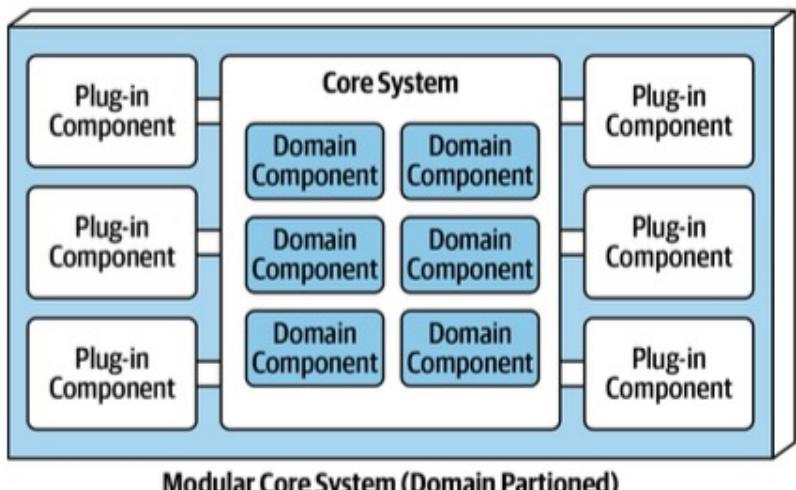


Plusieurs approches existent :

Question :

Comment le Core est implémenté ?

Utiliser une approche par couche



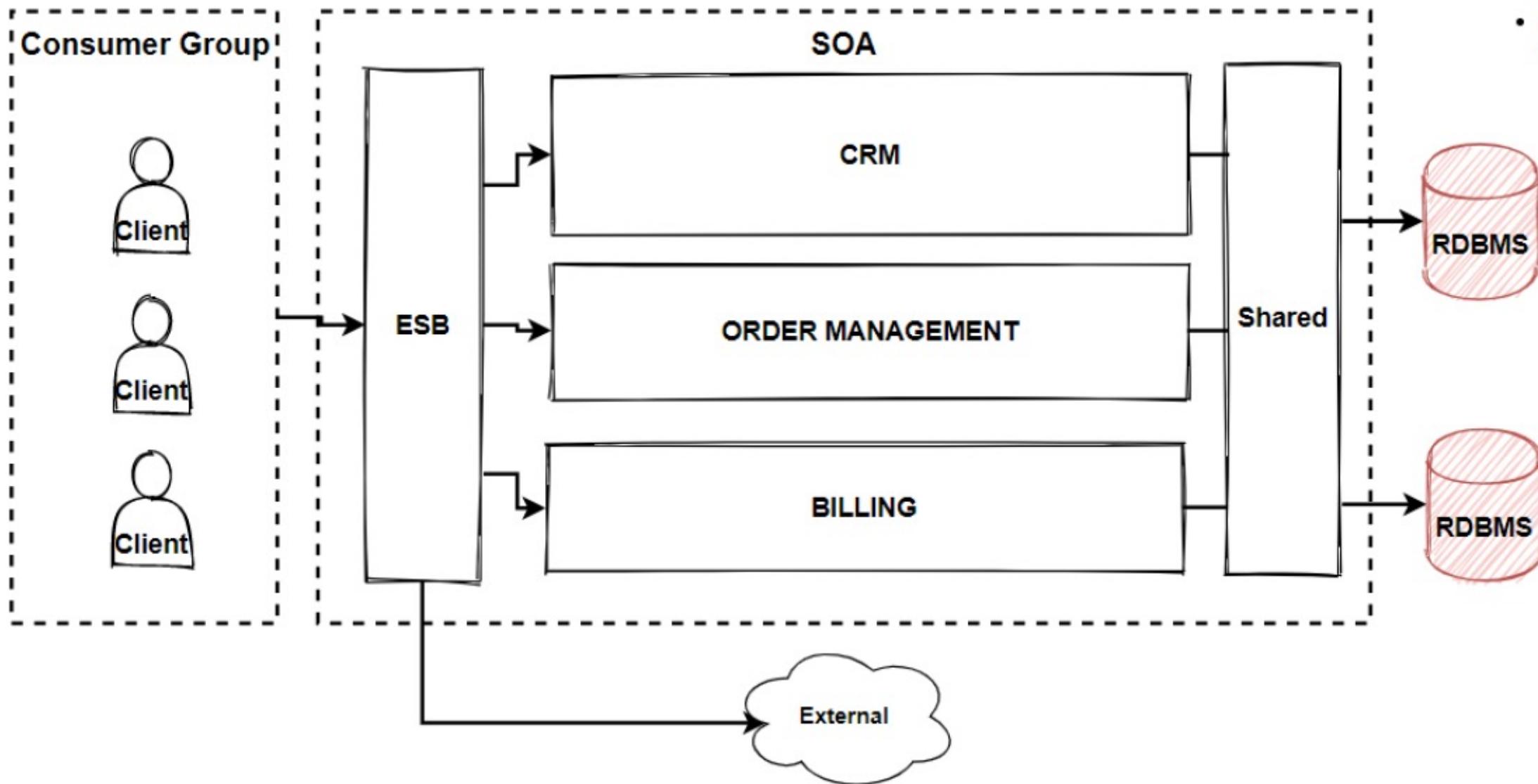
Question :

Comment le Core est implémenté ?

Utiliser une approche par domaine métier

Architecture Distribuée

#1 SOA

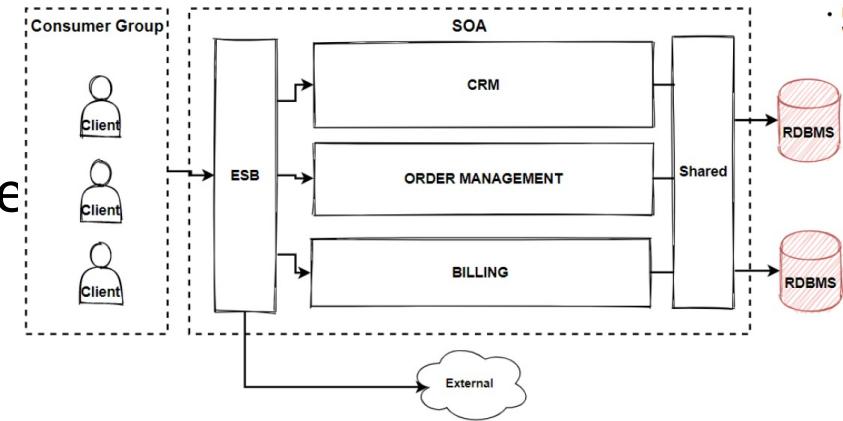


Définition SOA

Le SOA sépare les fonctions en unités distinctes (i.g. services), que les développeurs rendent accessibles sur un réseau afin de permettre aux utilisateurs de les combiner et de les réutiliser dans la production d'applications.

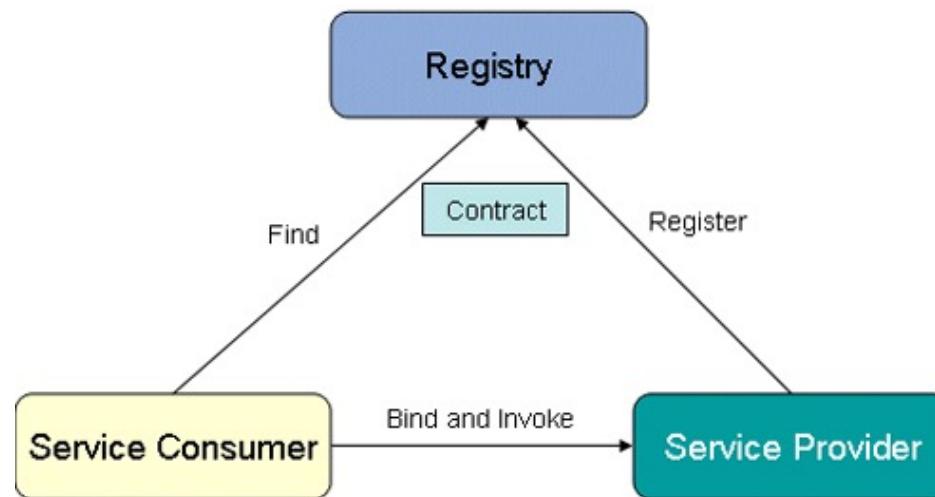
Un service :

- Est une représentation *logique* d'une activité commerciale ayant un résultat précis (e.g. vérifier le crédit d'un client, fournir des données météorologiques, consolider les rapports de forage).
- Est autonome
- Peut être composé d'autres services
- Est une “boîte noire” pour les consommateurs du service

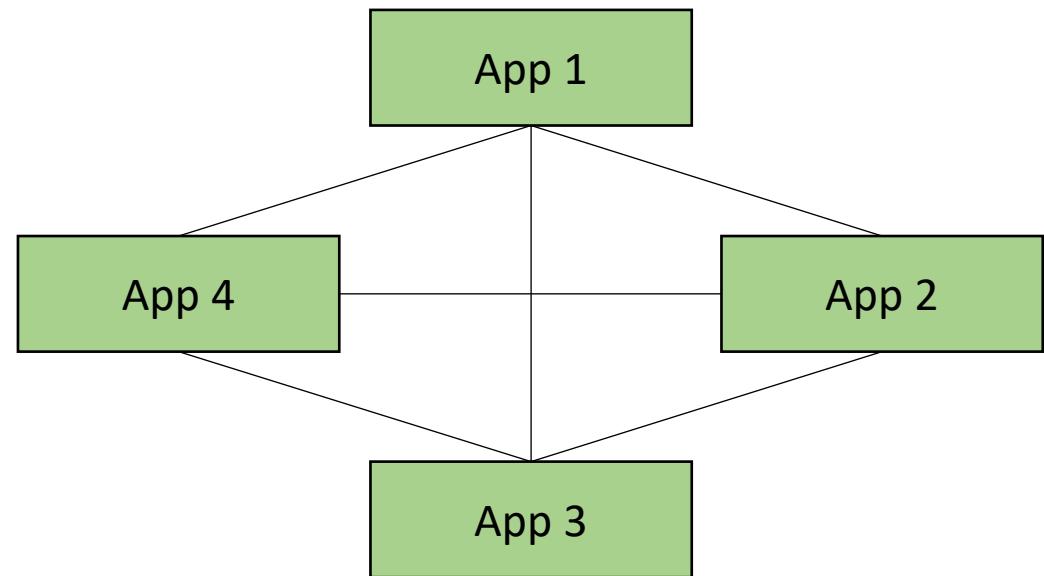


Composition : Registre/Broker

Un *registre de service*, ou broker, est un répertoire contenant les services disponibles, accessible via le réseau. Il stocke les contrats des services émis par les fournisseurs.

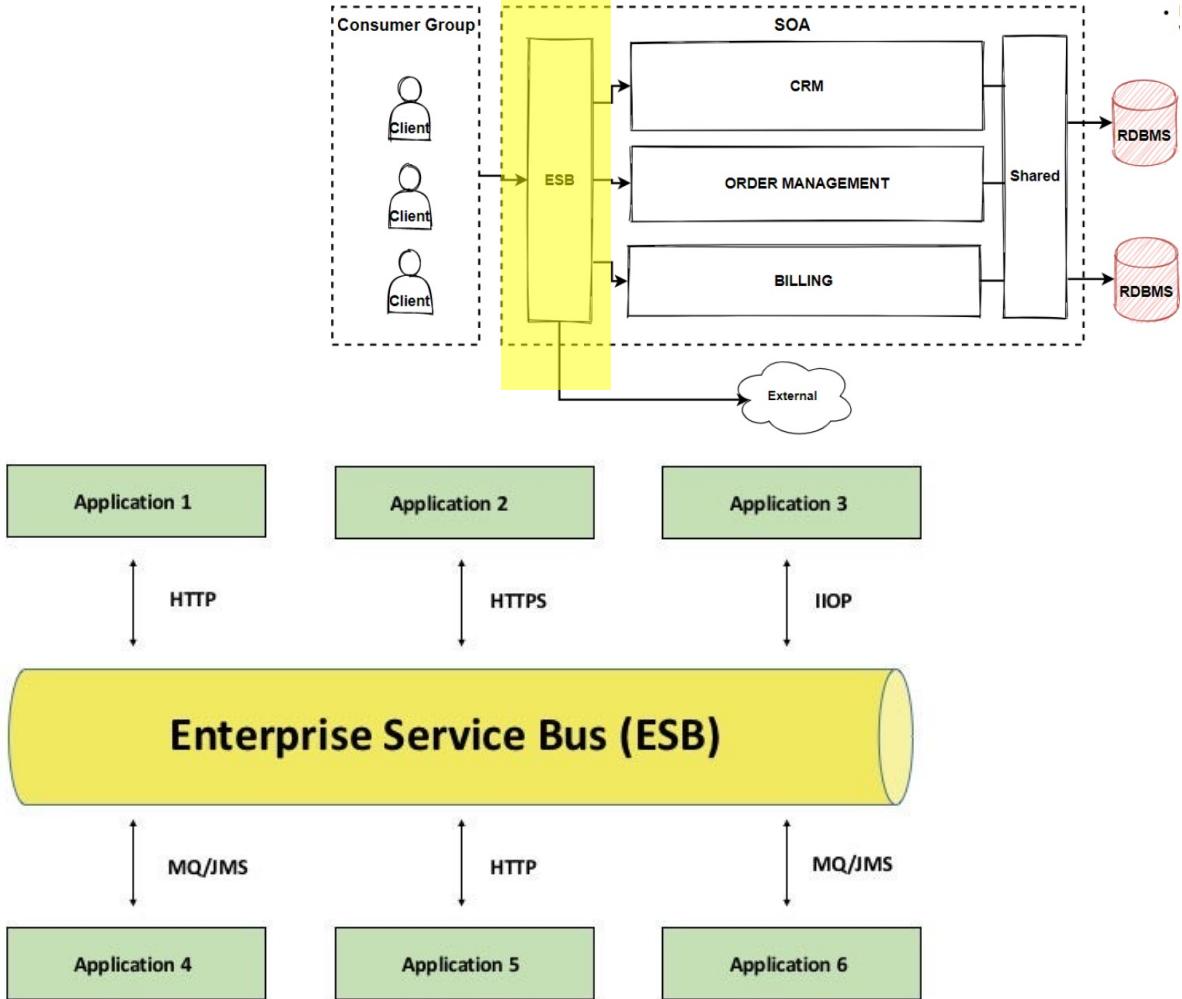


Composition : ESB



Eviter communication point-à-point, sinon
l'app1 devra :

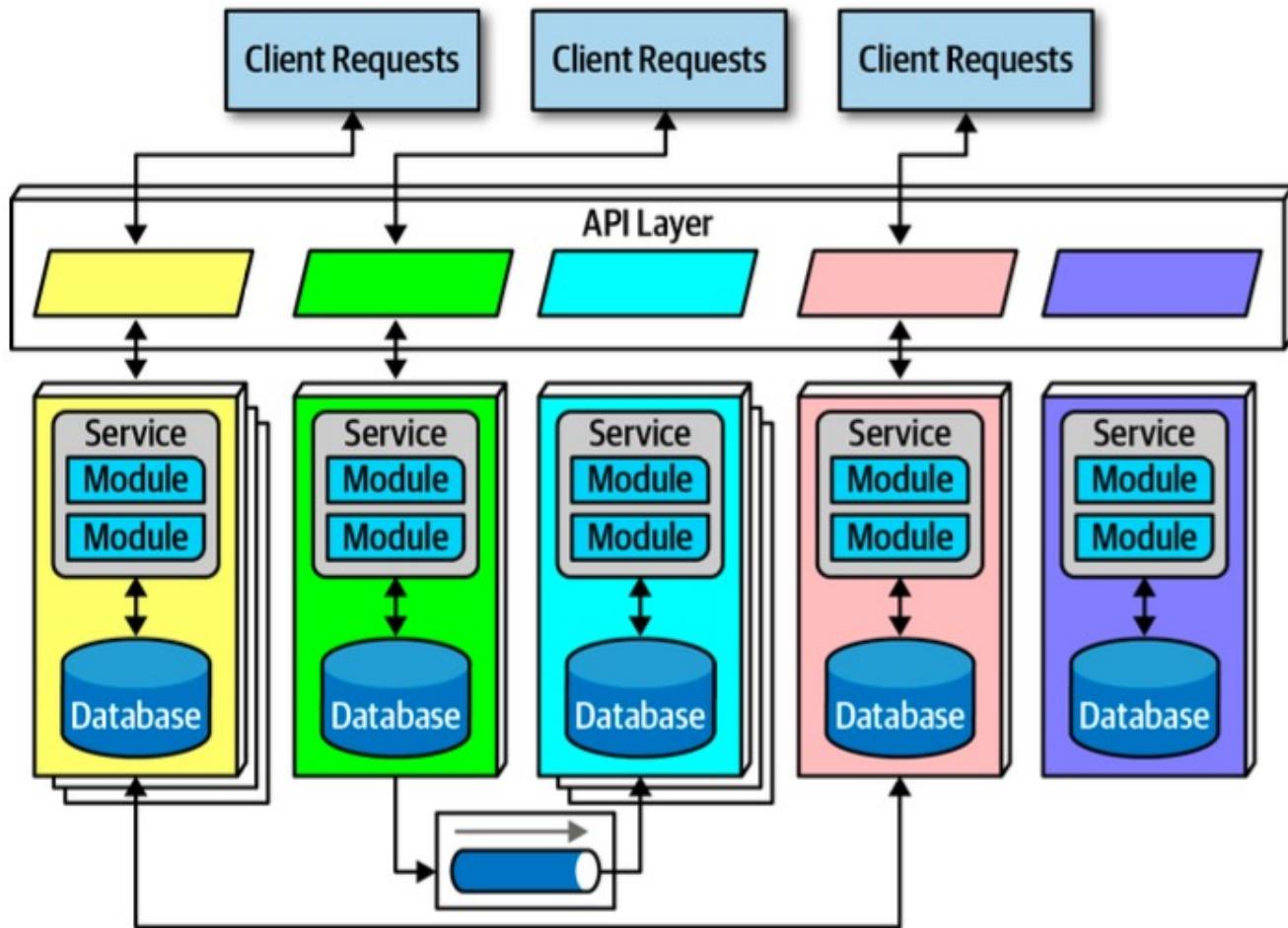
- Gérer le protocole http
- Gérer le protocole https (app2)
- Gérer le protocole IIOP (app3)
- Gérer le protocole JMS (app4)



On centralise tout, l'ESB lui gère les protocoles et fait les traductions

Architecture Distribuée

#2 Microservices



- Plusieurs services indépendants avec pour chaqu'un
 - Leur propre BDD
 - Leur propre frontend
- Puis, ils communiquent entre eux pour répondre aux besoins métier

Les problèmes techniques : l'atomicité

L'atomicité est un principe trivial en architecture monolithique, mais compliqué à mettre en place dans une architecture distribuée

Question :

Pourquoi ?

Avec l'architecture microservices nous souhaitons un découplage fort, mais comment doit-on assurer les transactions qui se déroulent au sein de plusieurs services ?

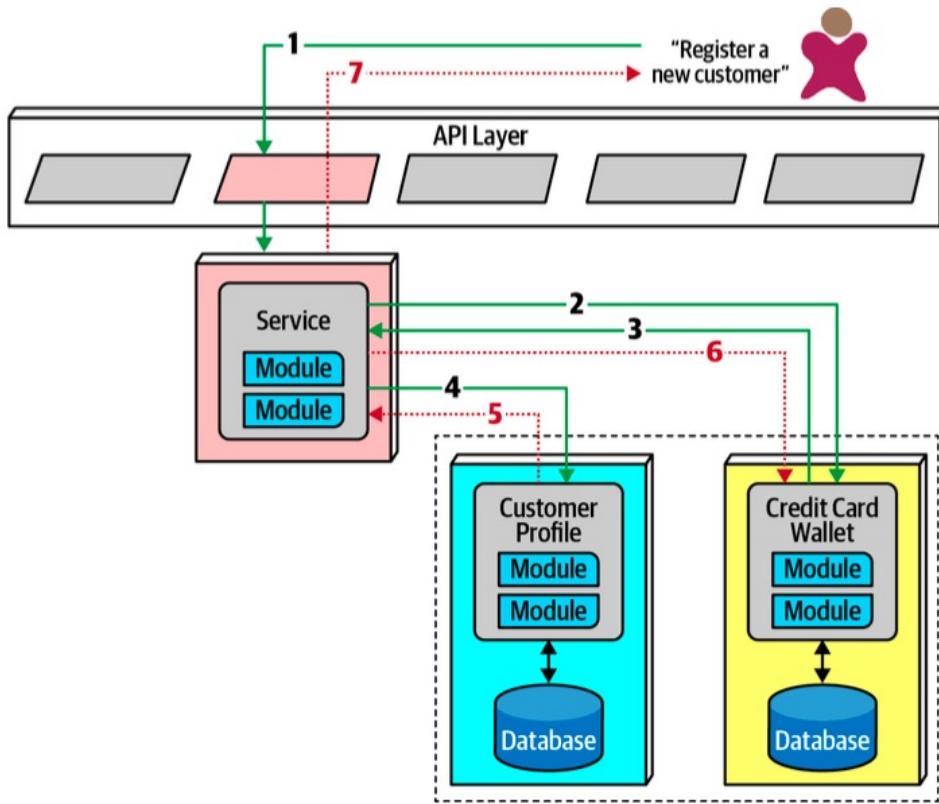
Transaction : comment faire ?

Question :

A votre avis comment gérer une transaction entre plusieurs microservices ?

- La théorie :
 - Le patron SAGA permet de gérer les transactions en utilisant une séquence de transactions locales de microservices. Chaque microservice possède sa propre base de données et peut gérer les transactions locales de manière atomique avec une cohérence stricte.
- Exemple : slide suivante

Transaction : comment faire ?



1. Le service médiateur reçoit une requête
2. Il effectue une requête vers le service CreditCardWaller
3. Qui lui retourne une réponse pour signifier que la demande est *enregistrée*
4. Le service médiateur envoie également une requête vers le service CustomerProfile
5. Mais une **erreur se produit**
6. Le service médiateur rollback donc la demande *enregistrée* en 3) pour revenir à un état stable
7. L'utilisateur est informé d'une erreur

Architecture Distribuée

#3 Event-driven

Objectif

Traiter des évènements de manière asynchrone

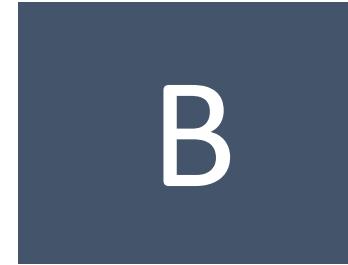
Vision traditionnelle



A veut appeler B



A envoie un msg à B



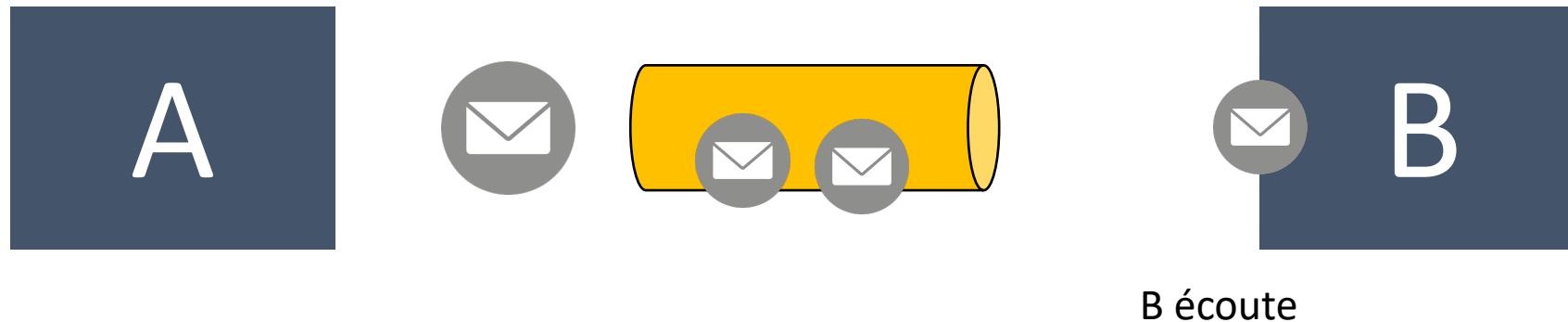
A envoie à la queue

B écoute la queue

B rend un service à A

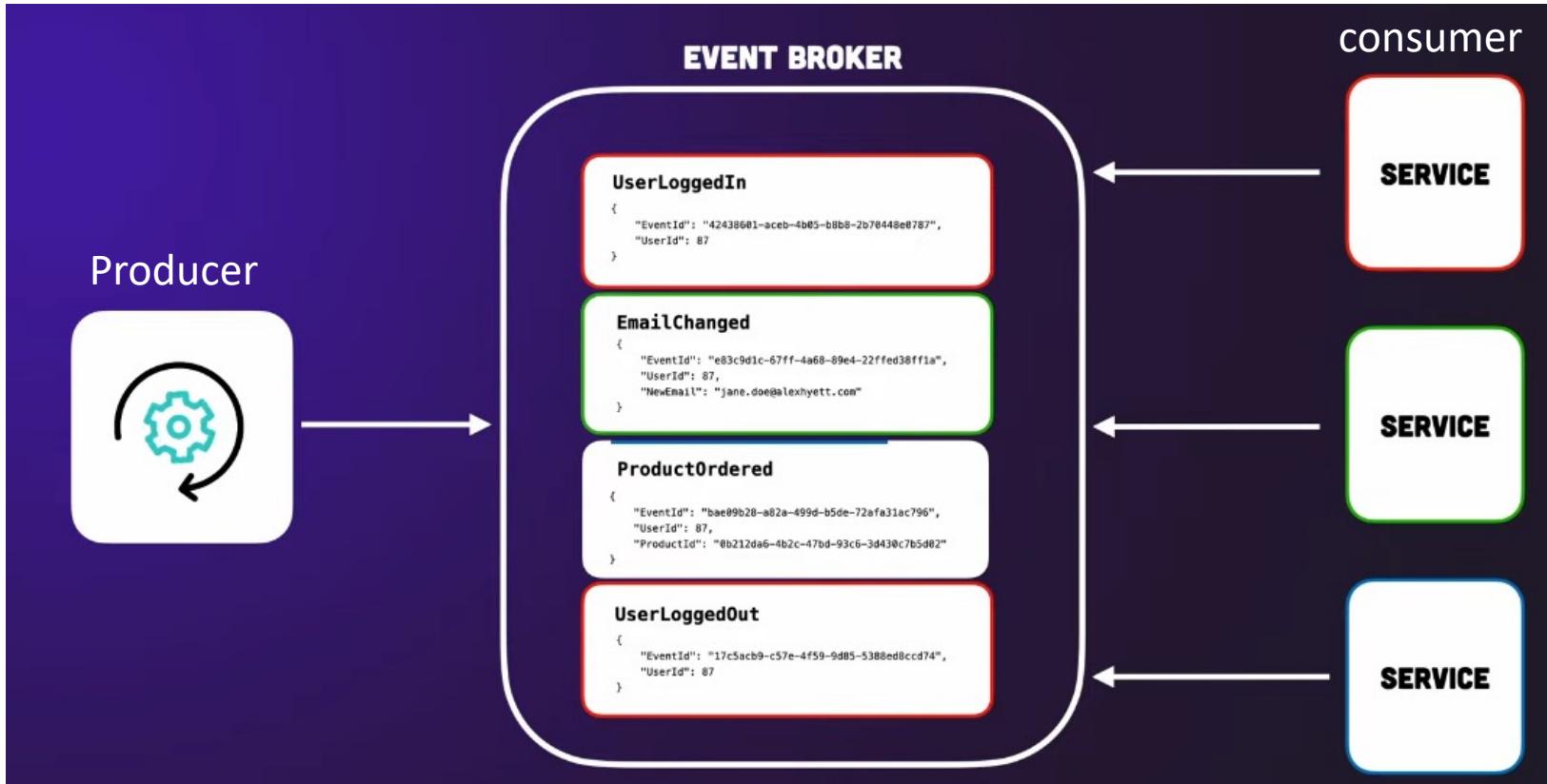
Vision évènement

B récupère les messages quand il est prêt à les traiter



A prévient par émission (et ne se préoccupe pas de B)

Procuder/Consumer



Chaque évènement dit à quel type d'évènement il s'abonne

A publie dans une queue plusieurs type de messages et puis les services récupèrent uniquement les leurs⁷

Quand l'utiliser

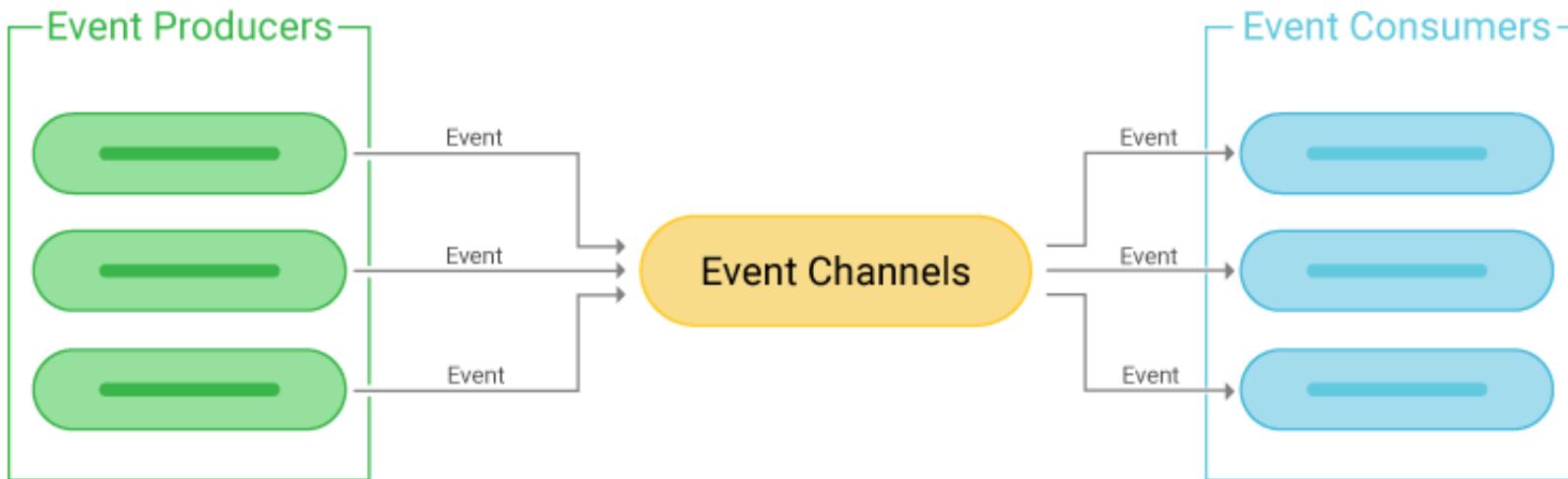
e.g : un système d'email après une commande

- Quand vous avez passé la commande
- Il n'est pas crucial que vous revenez le mail dans la seconde



- Le service d'email s'abonne au broker
- Le site de e-commerce publie sur le broker
- *Asynchrone, si le service d'email est en panne 10min on enverra le mail après.*

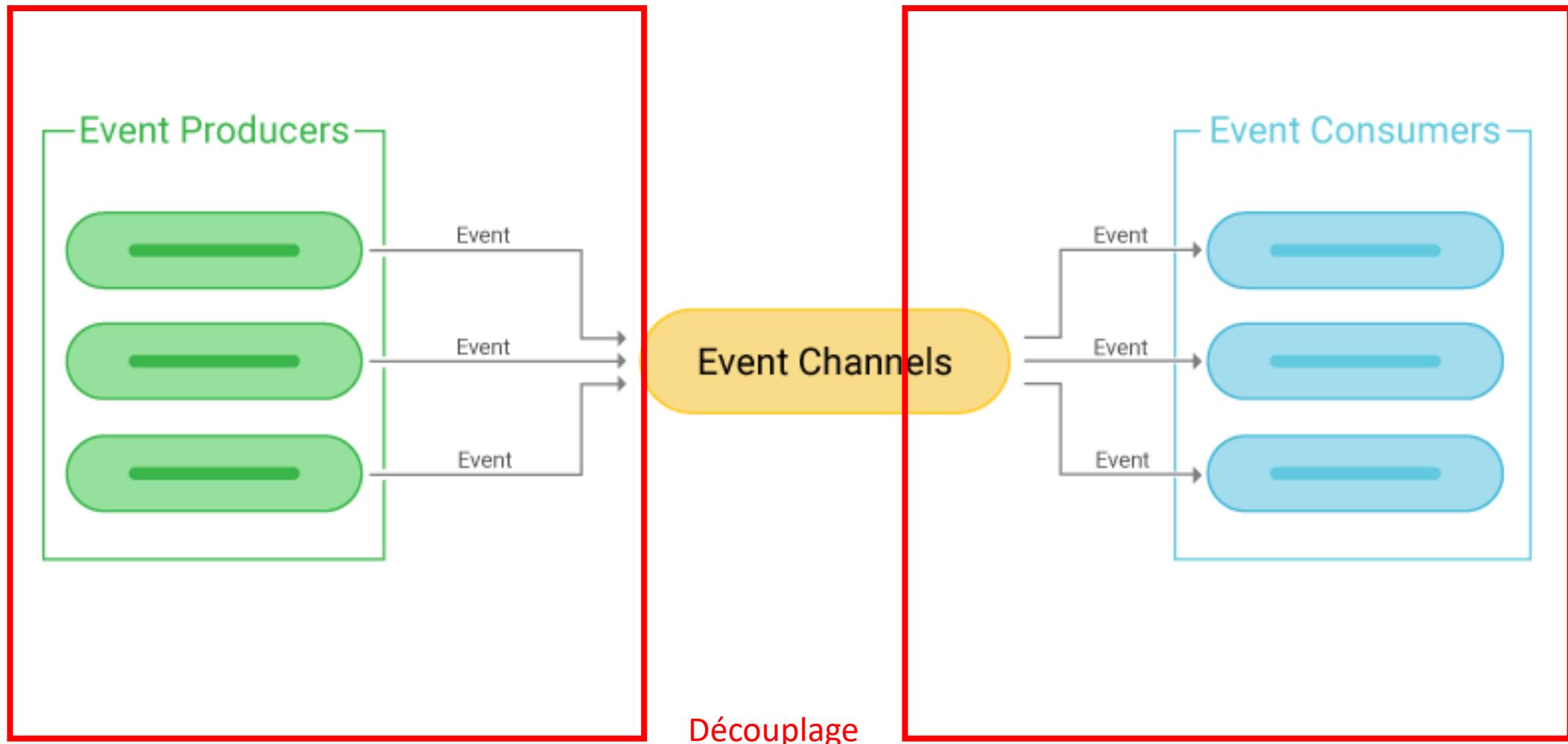
Avantages



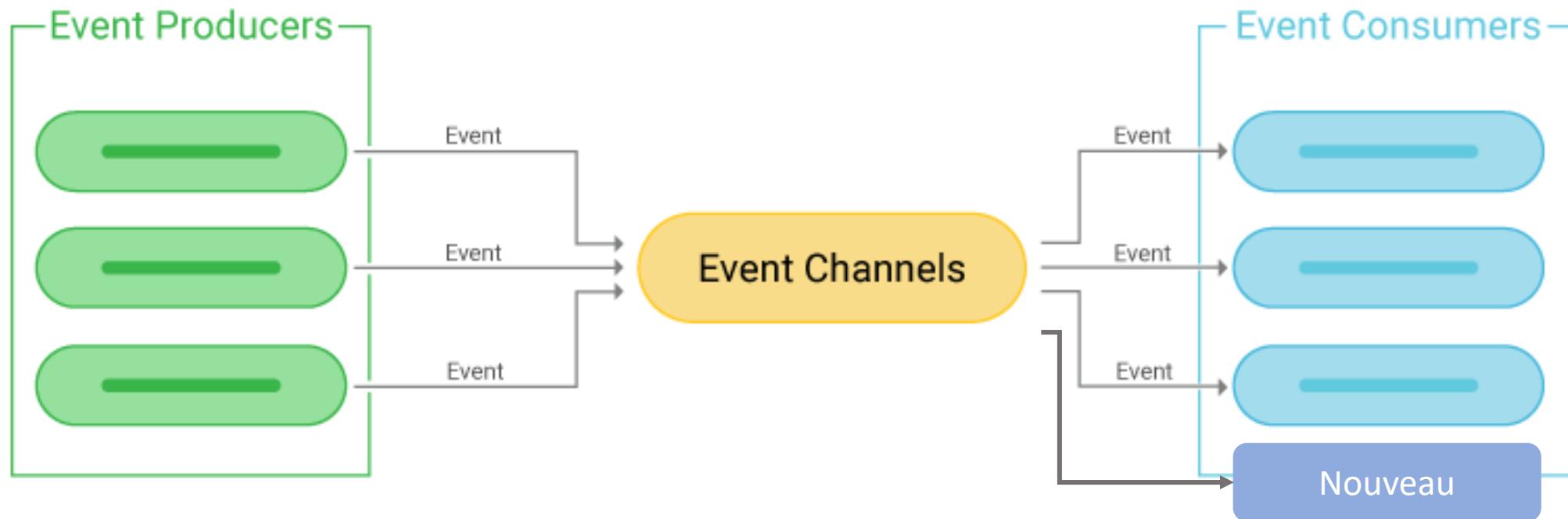
Question :

A votre avis, quels sont les avantages ?

Avantages

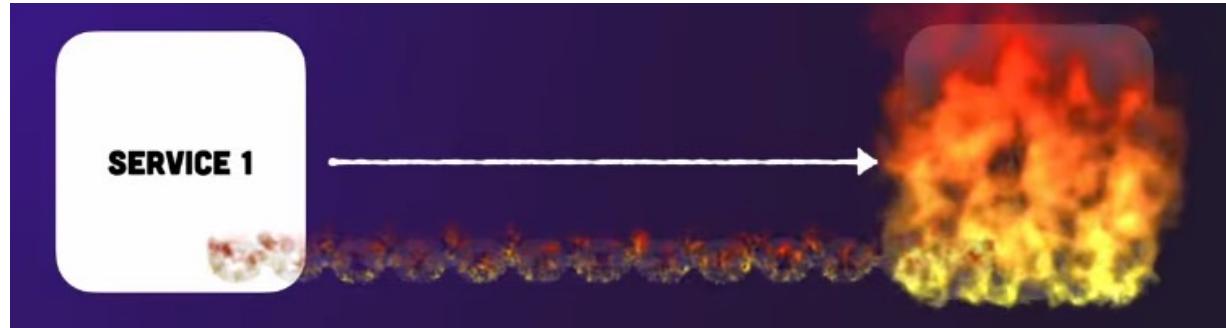


Avantages

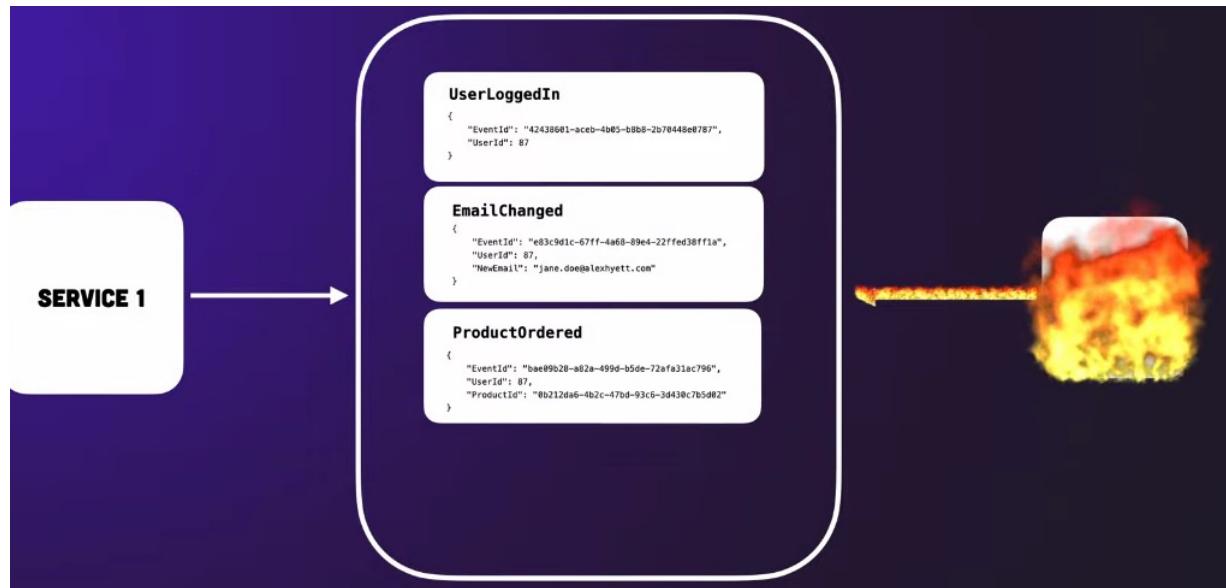


Ajout d'un nouveau consumer facilement

Avantages



Quand S1 dépend de S2 (mode synchrone)
Alors si S2 tombe S1 ne fonctionne plus



S2 offline, aucun problème
Le broker stocke les messages

Et quand S2 sera up il lire les messages conservés dans le broker