

# Architecture Logicielle

Adrien CAUBEL

Référence principale :

- Fundamentals of software architecture Mark Richard

# Infos complémentaires

- L'ensemble de ce cours est disponible sur  
<https://softwarearchitecture.fr/>

*Le site est toujours en cours de construction*

# Objectifs pédagogiques

# Objectifs pédagogiques

- Acquérir du vocabulaire
- Présentation des concepts fondamentaux « de bonne conception »
- Présentation des **styles architecturaux** communs

Question :

A votre avis que signifie « styles architecturaux »

Ce cours s'inscrit dans les pratiques d'intégration

# Pourquoi connaître du vocabulaire ?

« One of the valuable features of patterns work is that it develops a vocabulary with which we can talk about how to do things. »

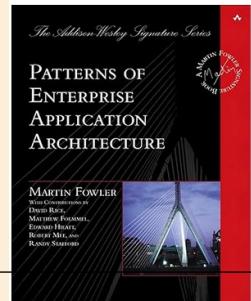
# Pourquoi connaitre les architectures ?



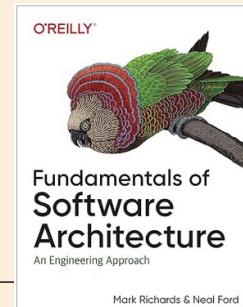
Construire Logiciel == construire sur de l'existant, alors il vaut mieux que la base soit solide

# Pourquoi ce cours en parcours D?

- Un fort lien avec l'agilité
- Concepts intemporels



2003



2020

Toujours  
d'actualité

2040

- Culture générale; pouvoir parler avec des devs

# What we will learn ?

# Table des matières

1	Caractéristiques Architecturales
2	Concepts fondamentaux
3	Styles architecturaux
4	Patrons architecturaux

# Caractéristiques architecturales

1

# 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

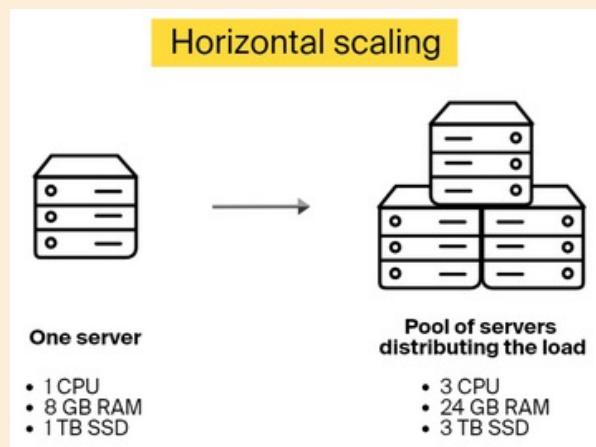
# Scalabilité

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

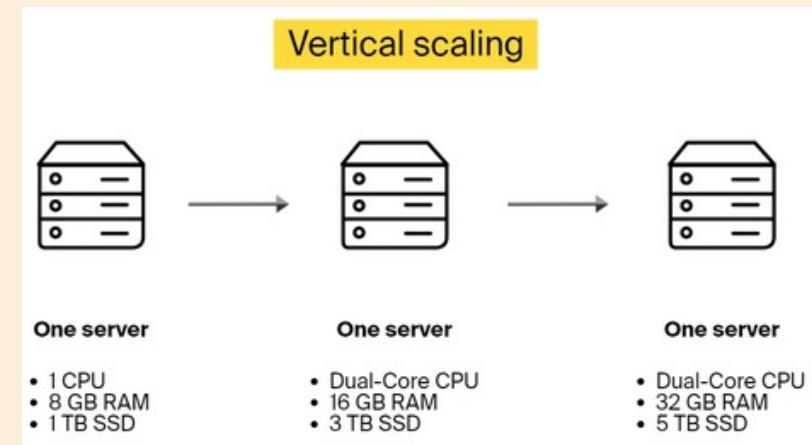
Question :

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.

# Low Coupling, High Cohesion

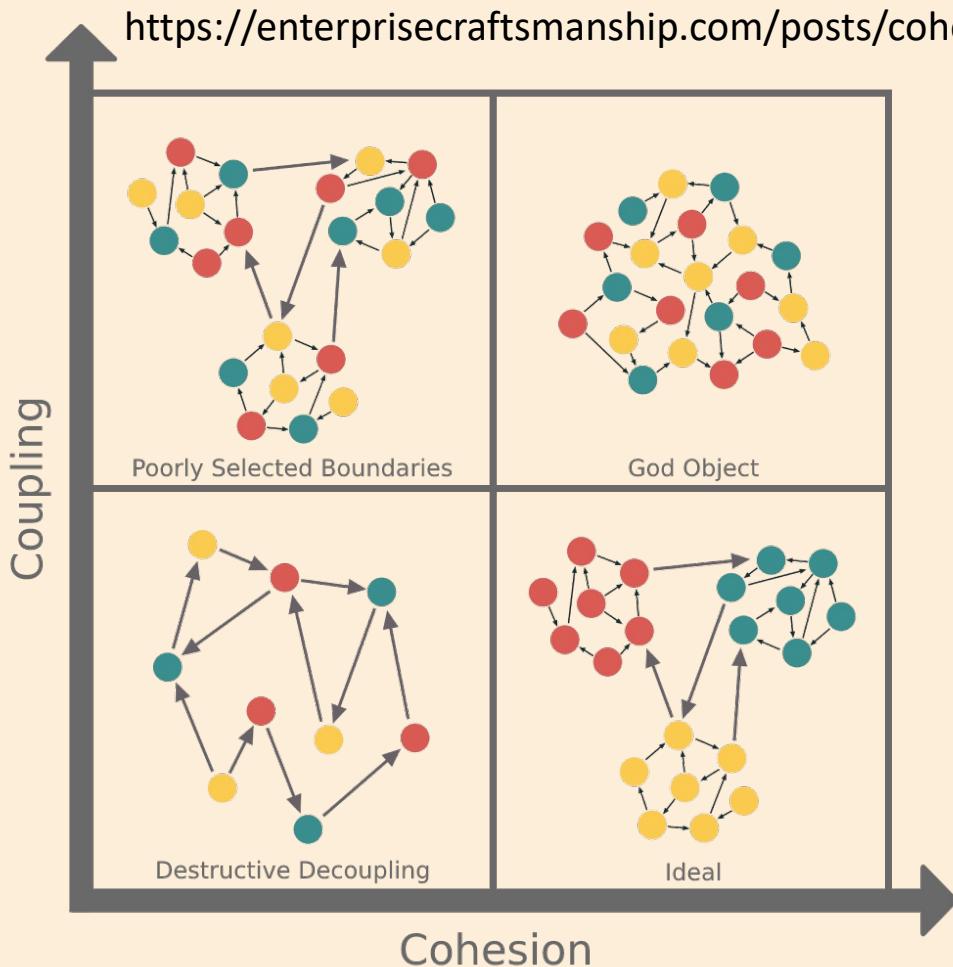
Concevoir des composants autonomes, indépendants et dotés d'un objectif unique et bien défini.

Question :

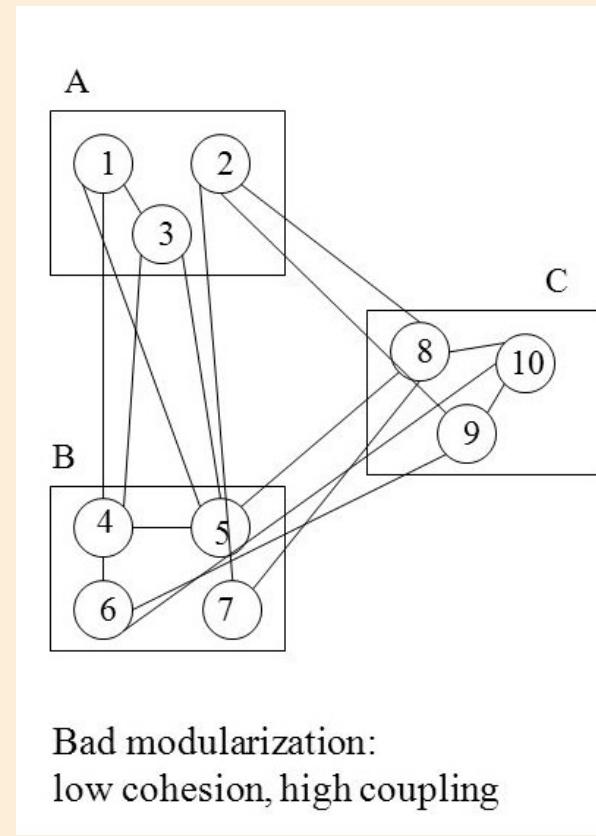
Que cela peut-il signifier (couplage et cohésion) ?

- **Couplage faible** : Les modules doivent être aussi indépendants que possible des autres modules, de sorte que les modifications apportées au module n'aient pas d'impact important sur les autres modules.
- **Forte cohésion** : La cohésion fait référence à la manière dont les éléments d'un module s'intègrent les uns aux autres. Les codes apparentés doivent être proches les uns des autres afin d'assurer une grande cohésion.

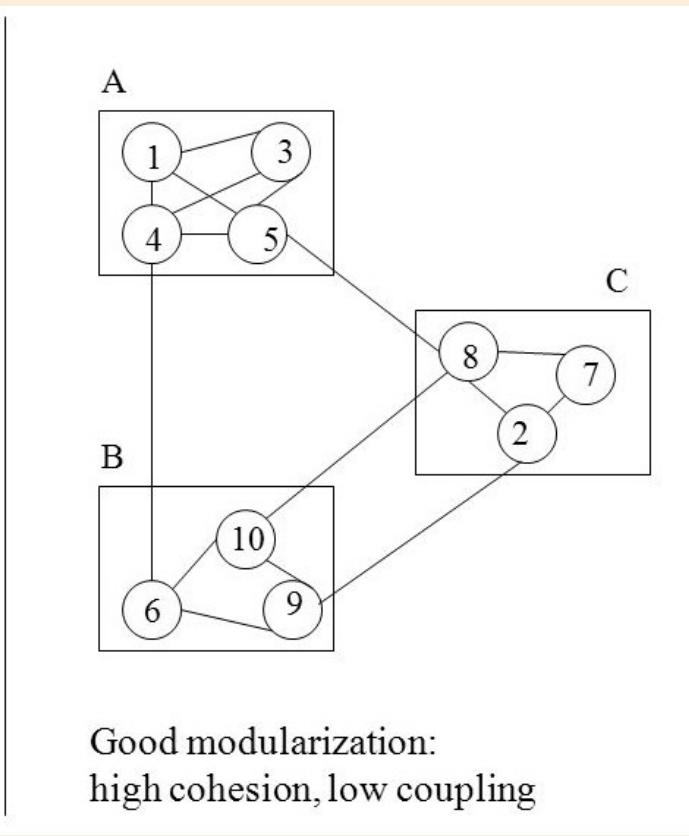
# Low Coupling, High Cohesion



Here, circles of the same color represent pieces of the code base related to each other.



Bad modularization:  
low cohesion, high coupling

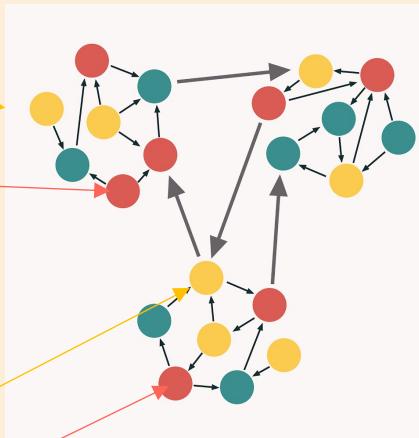


Good modularization:  
high cohesion, low coupling

# Low Coupling, High Cohesion (package)

► C# DomainModel
► Properties
► References
► Entities
► Order.cs
► OrderLine.cs
► Product.cs
► User.cs
► Factories
► ProductFactory.cs
► UserFactory.cs
► Repositories
► OrderRepository.cs
► ProductRepository.cs
► UserRepository.cs
► Services
► ProductService.cs

Bien organisé, mais manque de cohésion

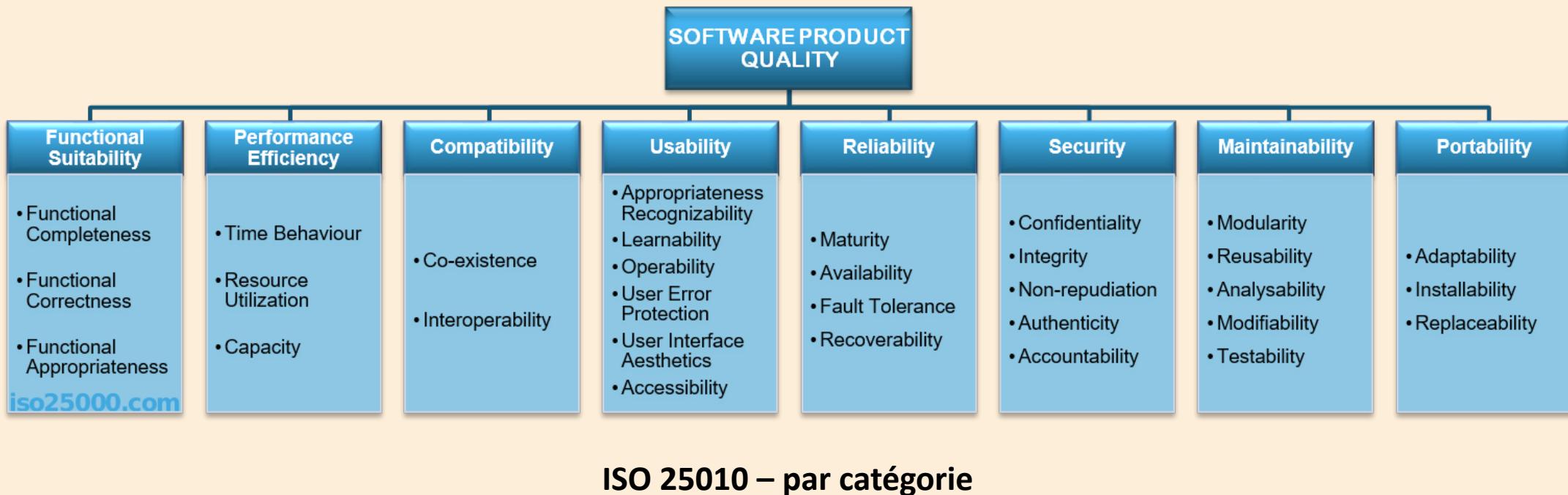


Il faut que les « frontières » représentent la sémantique

=> Forte cohésion

► C# DomainModel
► Properties
► References
► Orders
► Order.cs
► OrderLine.cs
► OrderRepository.cs
► Products
► Product.cs
► ProductFactory.cs
► ProductRepository.cs
► ProductService.cs
► Users
► User.cs
► UserFactory.cs
► UserRepository.cs

# Beaucoup d'autres



ISO 25010 – par catégorie

# Concepts fondamentaux

2

# Objectif de la section

- Revoir (ou découvrir) des principes de conception élémentaires

Architecture & Intégration

Principes de conception élémentaire

Variables, POO, etc ...



# Plan concept fondamentaux

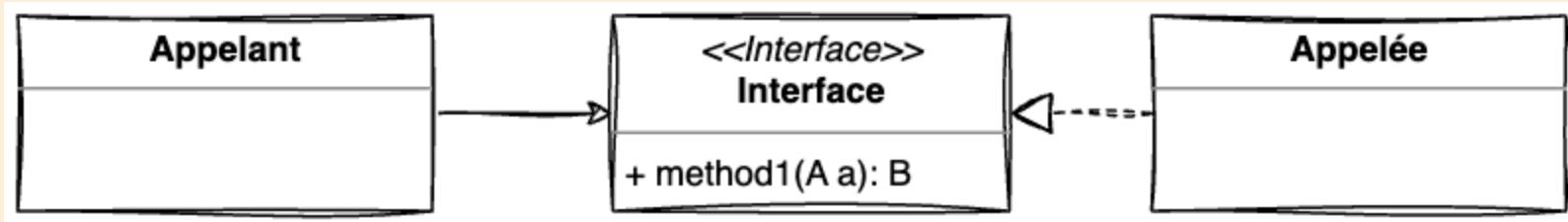
1. Les interfaces
2. L'inversion de dépendances
3. Les *Services*
4. Les *Modules*

# Interface

Question :

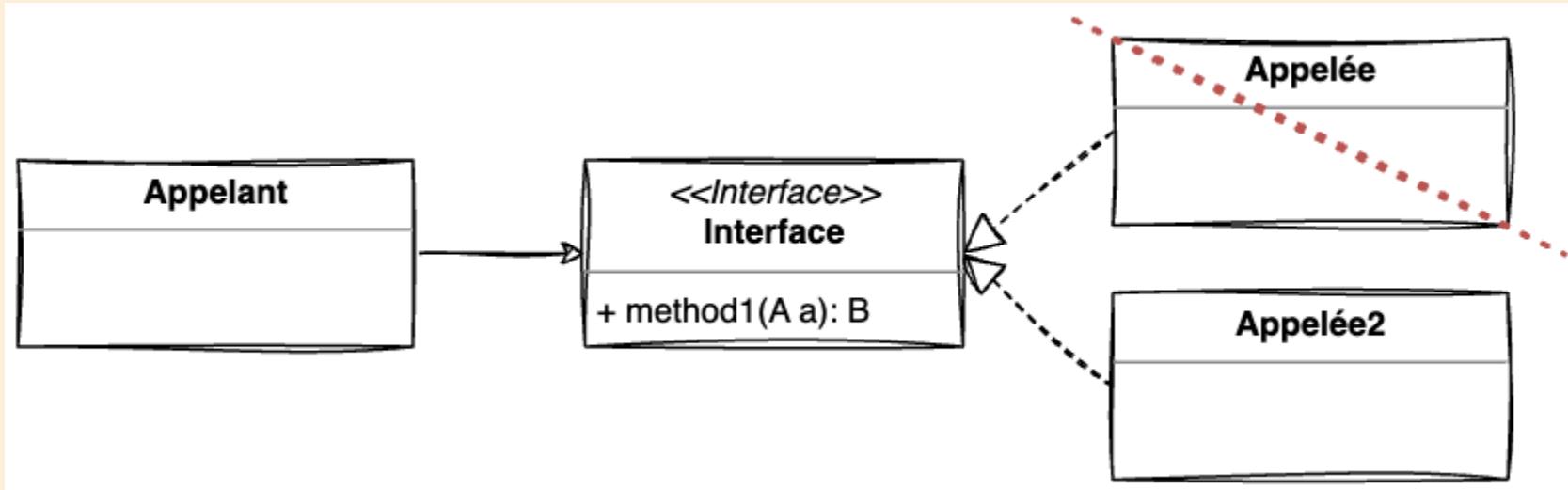
Quel est le rôle d'une interface ?

# Interface



- Elle définit les méthodes (i.g. service) qui vont pouvoir être appelées
- Elle définit les informations que doit fournir l'appelant (i.g paramètre de la méthode)
- Elle définit les informations que l'appelant va obtenir (i.g type de retour)

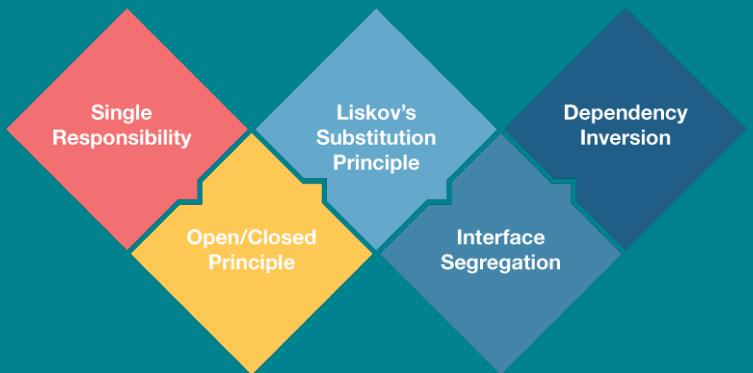
# Changer l'implémentation



Note :

Cycle build -> test -> déploy

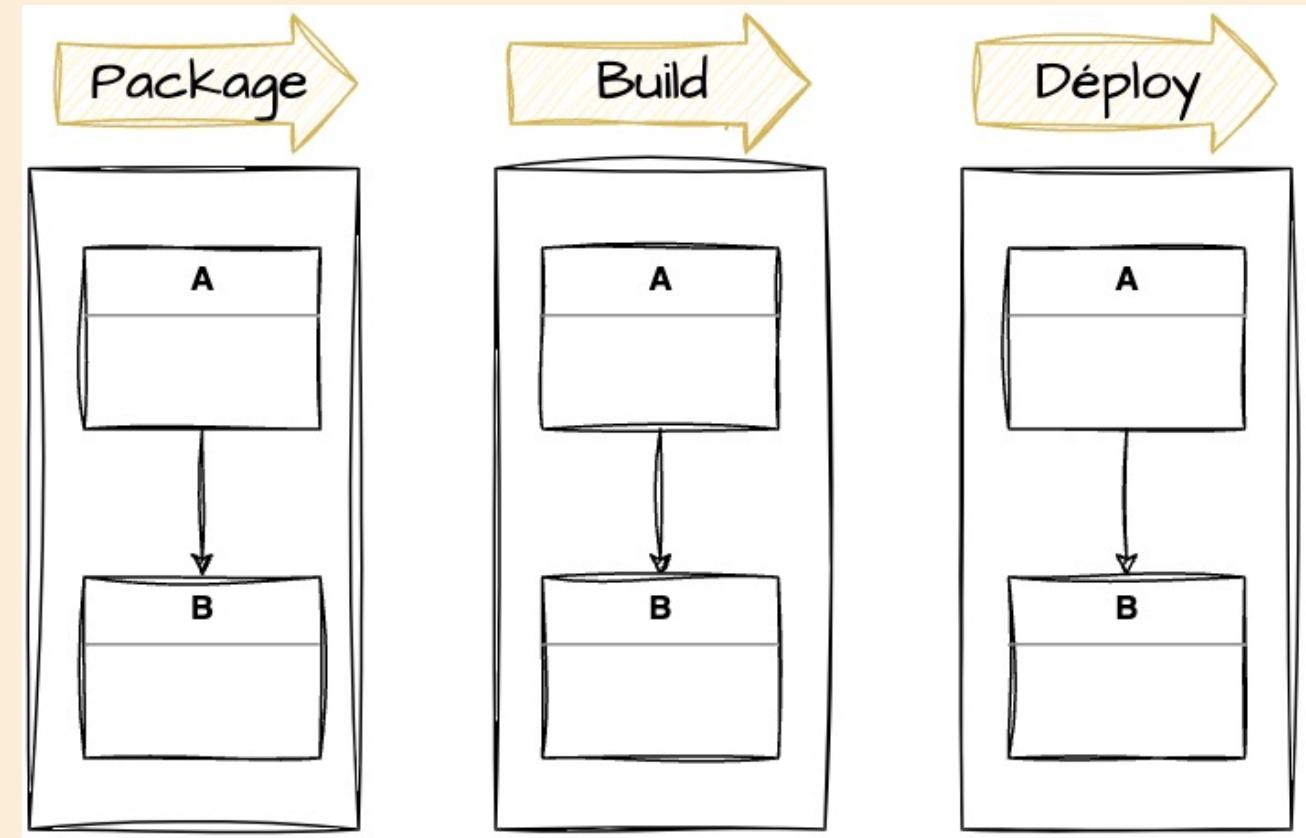
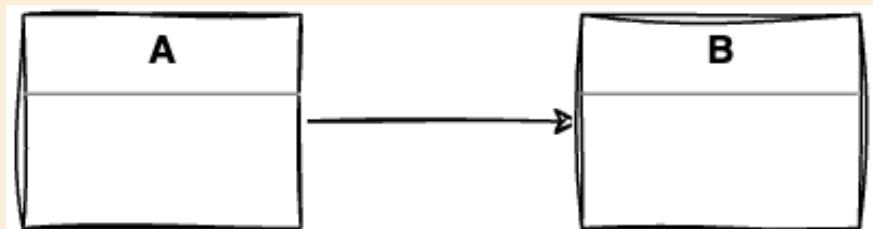
# Inversion de dépendance



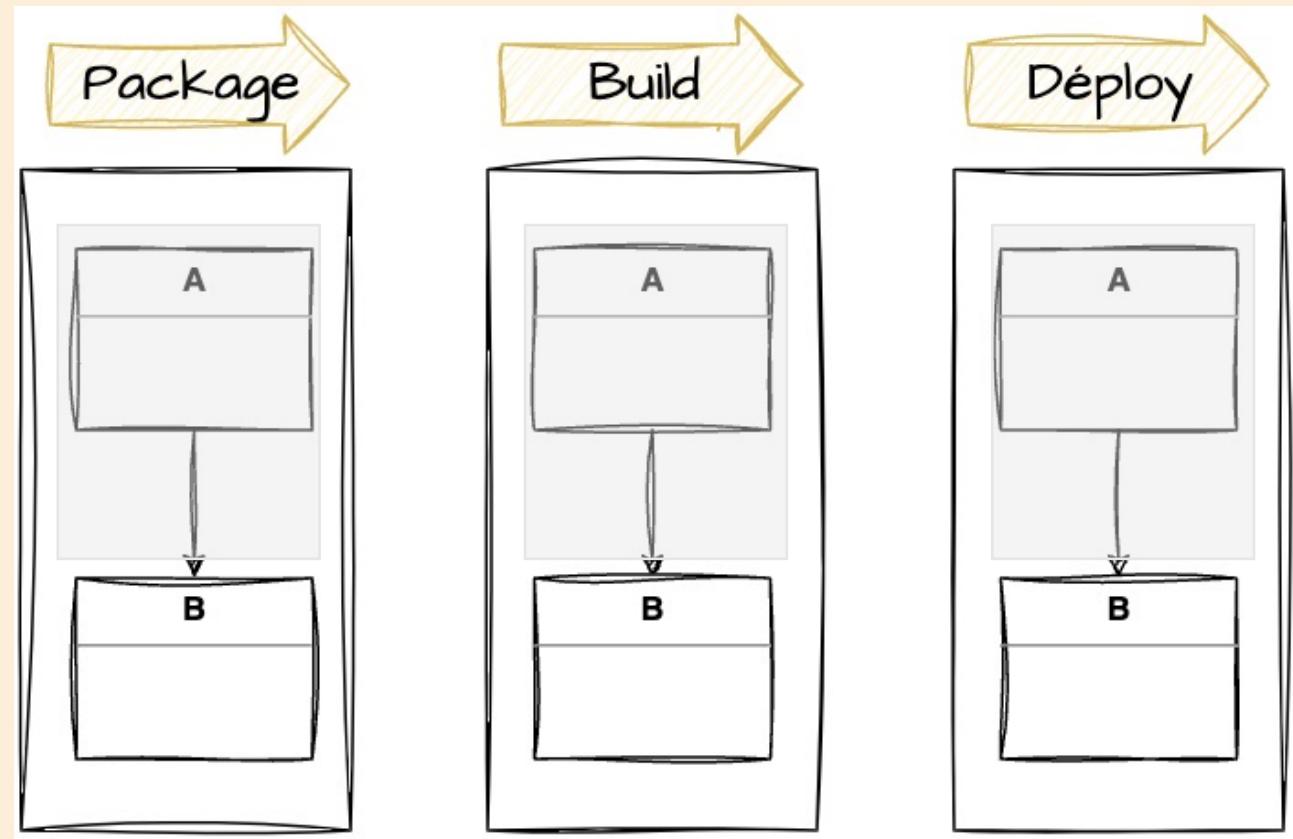
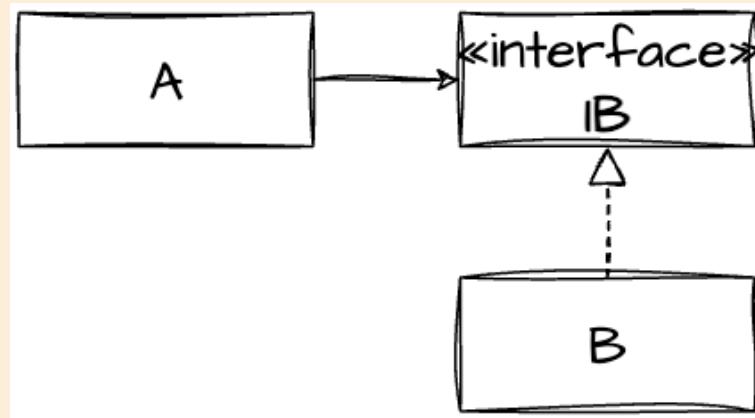
Question :

Que pouvez me dire de ce principe ?

# Sans Inversion de dépendance

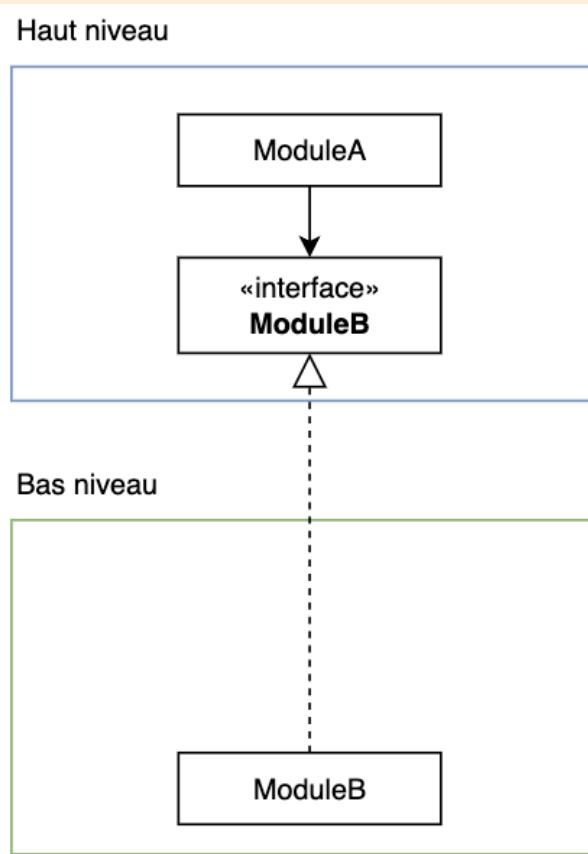
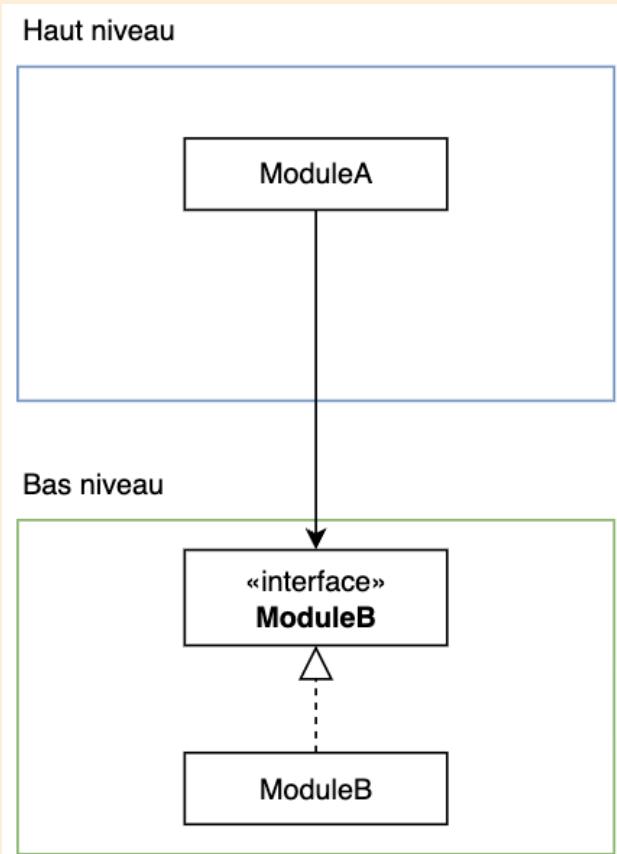


# Avec Inversion de dépendance



- A ne dépend plus de B mais de son interface IB
- Donc si B est modifiée alors seulement B sera packagée, build et déployée chez le client

# Amélioration



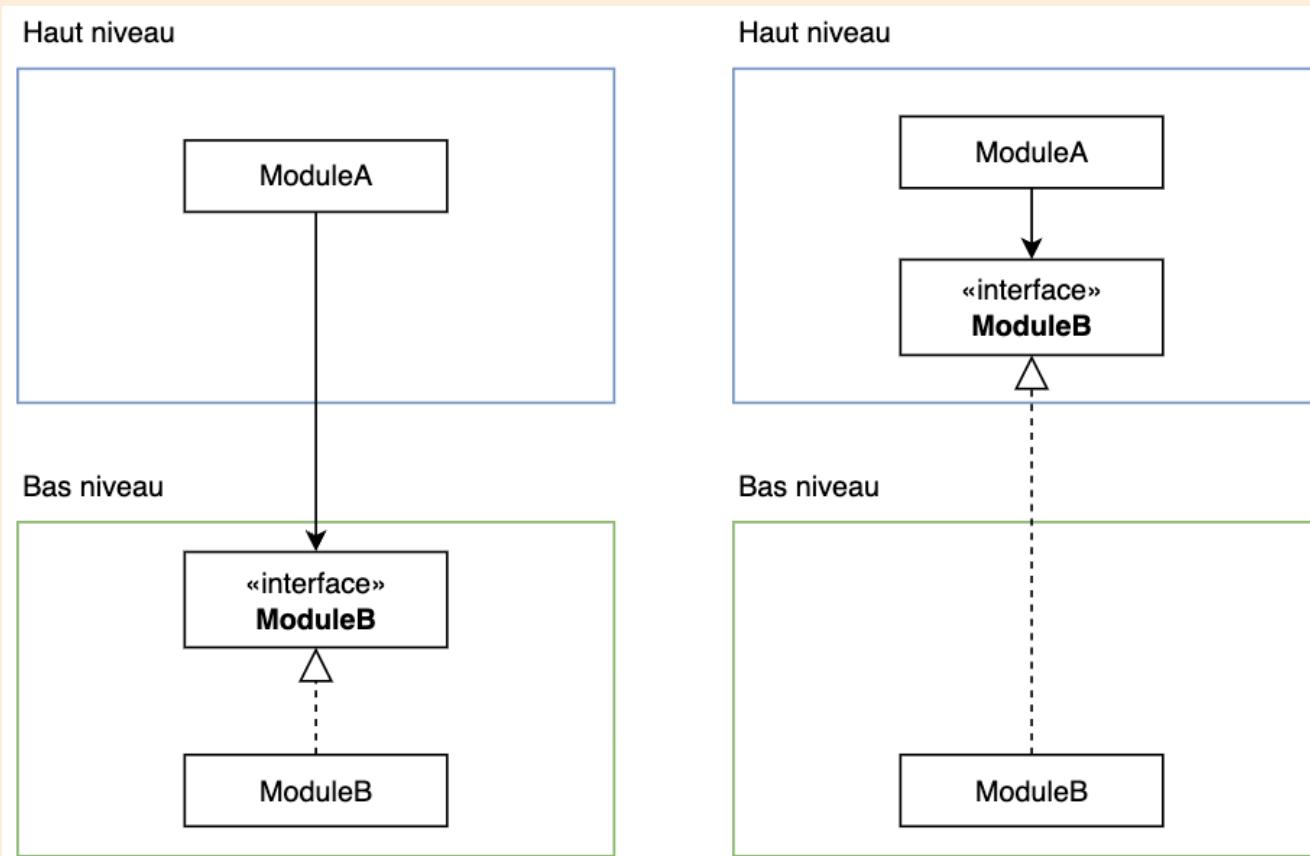
## A droite :

- C'est le module de bas niveau qui définit comment on interagit avec lui
- => c'est le module de bas niveau qui dirige

## A gauche

- On change « la phrase »
- Le module de haut niveau dit les fonctionnalités requises par chaque implémentation
- => c'est le module de haut niveau qui dirige

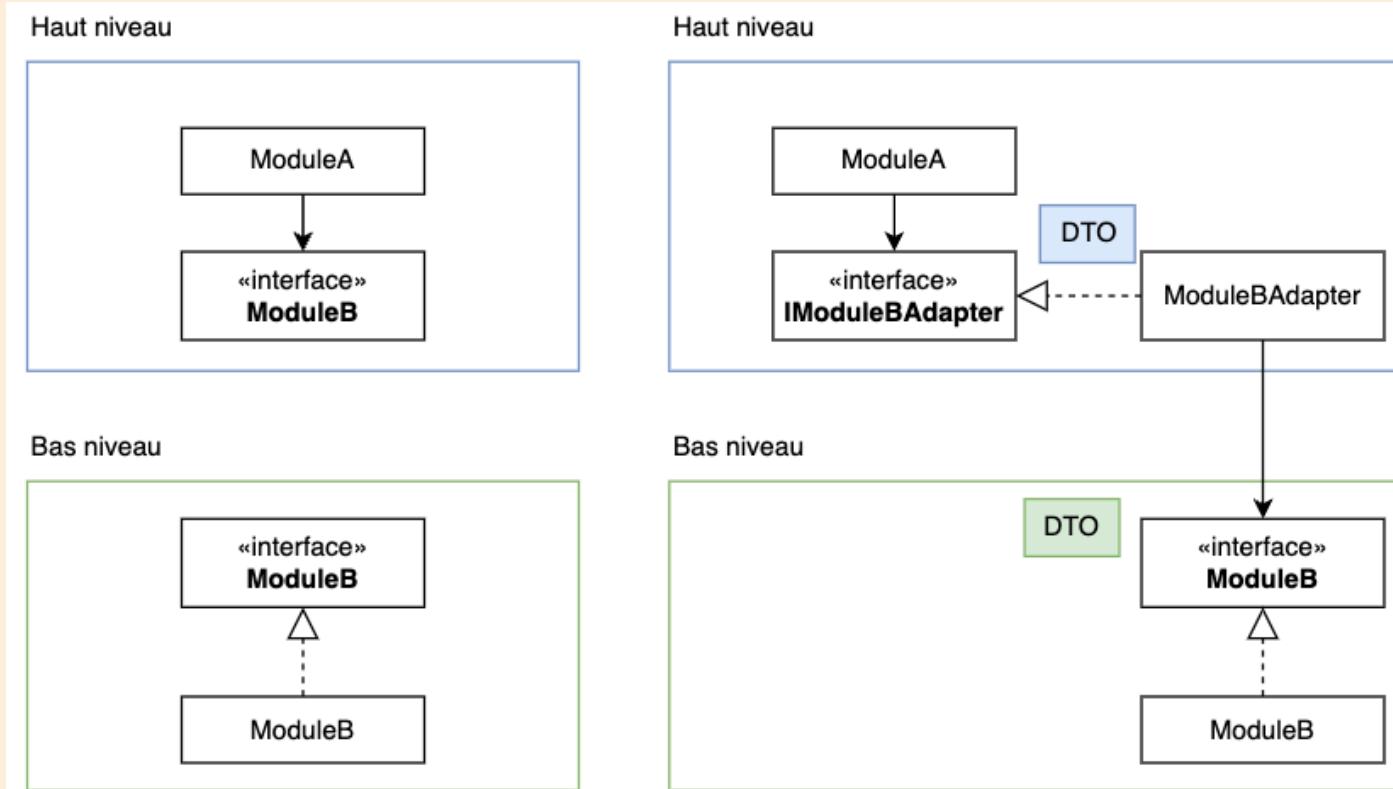
# Amélioration; MAIS ...



**Avec cette conception :**

- Il est impossible de réutiliser le module de bas niveau dans un autre contexte
- En effet, c'est le module de haut niveau qui dirige et le module de bas niveau subit l'interface

# Amélioration; DONC ...



## On rajoute une interface

- Les deux modules peuvent maintenant être réutilisés dans n'importe quel contexte

## Faire le lien

- Pour faire le lien entre les deux modules on utilise le **patron adaptateur** et la notion de **DTO**

- L'adaptateur implémente l'interface du module de haut niveau
- L'adaptateur utilise l'interface du module de bas niveau
- L'adaptateur permet aussi de convertir les structuteurs de données entre les deux modules.

# Exercice : envoyer un mail

## Exercice :

Vous souhaitez que votre application envoie un mail

- Vous utilisez la librairie JavaSimpleMail
- Les infos en interne sont : destinataire, sujet et contenu (et pas le protocole)

Dessiner la conception au tableau

IJavaSimpleMail

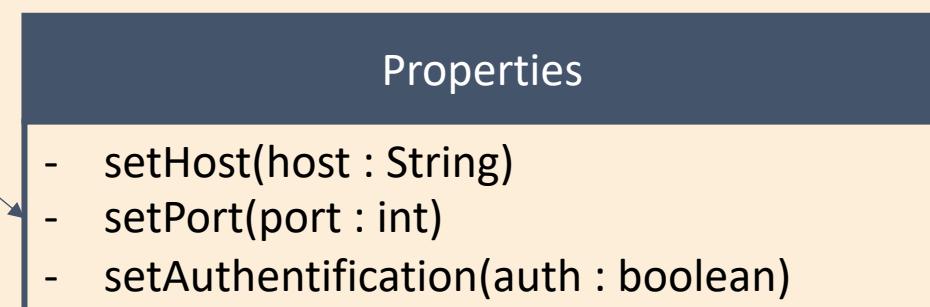
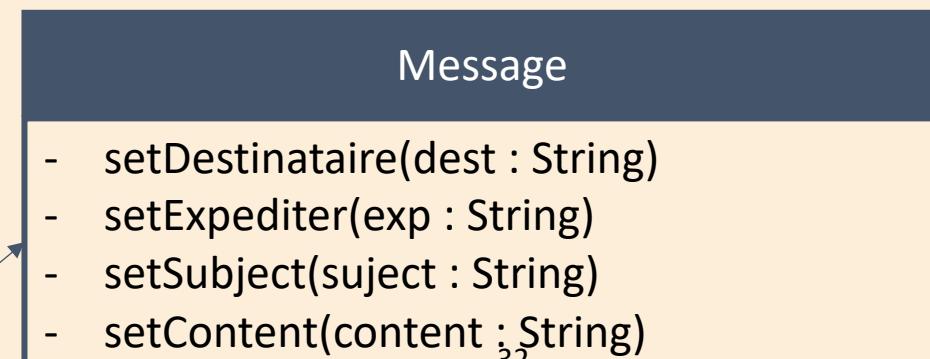
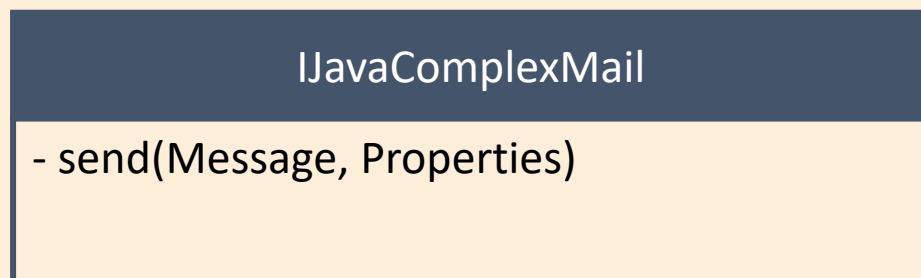
- send(from, to, subject, content, protocole)

# Exercice : envoyer un mail

## Exercice :

On rajoute de la complexité :

- On souhaite utiliser JavaComplexMail
- **On ne change pas l'interface Adaptateur**



# Un Service

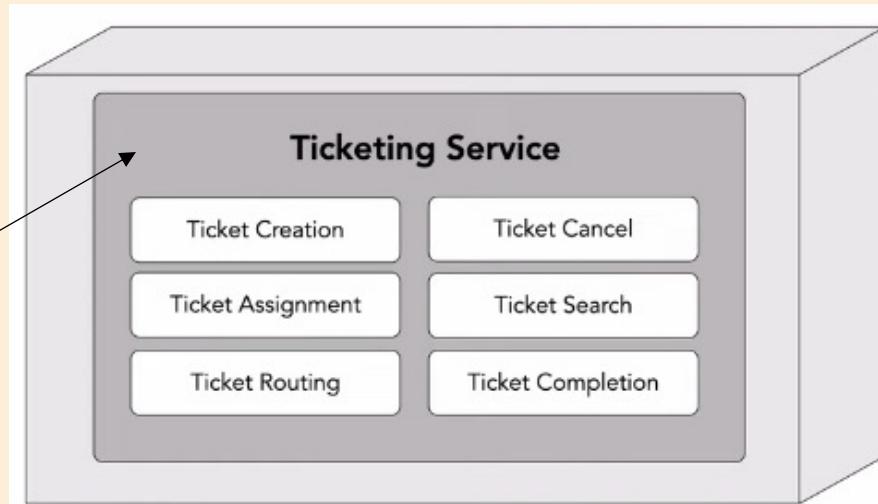
# Définition Service

Un service est une unité déployable qui accomplit une activité métier ou d'infrastructure

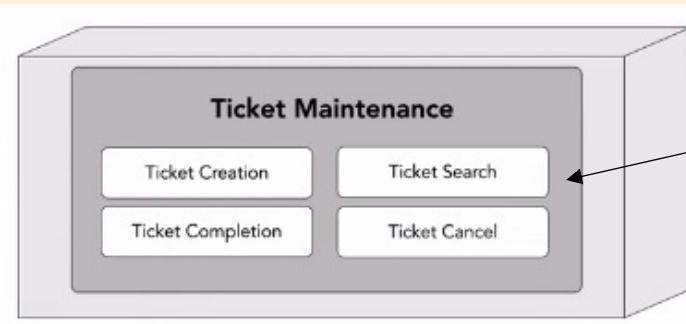
# Granularité d'un service ?

- C'est la question !!!

Service Ticket



Service Ticket Maint.



Service Ticket Routing



Question :

On fait quoi ? Un gros service ou plusieurs petits ?

# La réponse ... ça dépend !

## Architecture microservices

- Une fonctionnalité métier
- Plusieurs centaines de services
- Forte interconnexion

## Architecture service-based

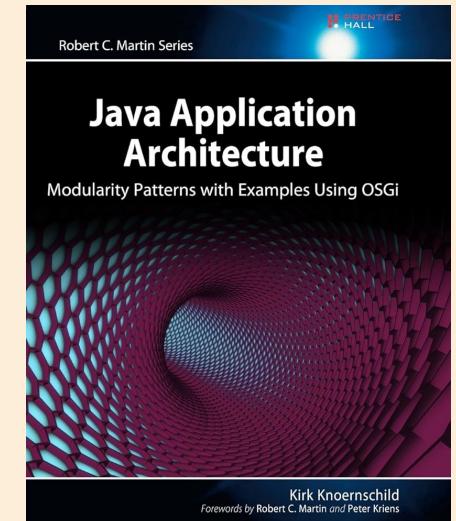
- Plusieurs fonctionnalités métiers
- De 3 à 12 services
- Eviter les interconnexions

# Un Module

# Définition module

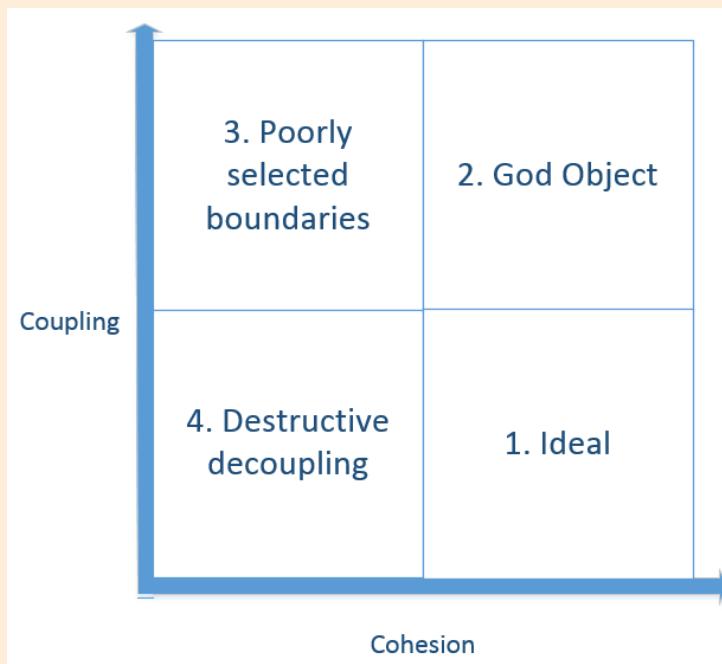
Un module logiciel est une unité logicielle déployable, « manageable », réutilisable, composable et stateless qui fournit une interface concise au client

A noter que la modularité participe à la réussite d'un logiciel dans le temps et est un sujet très complexe

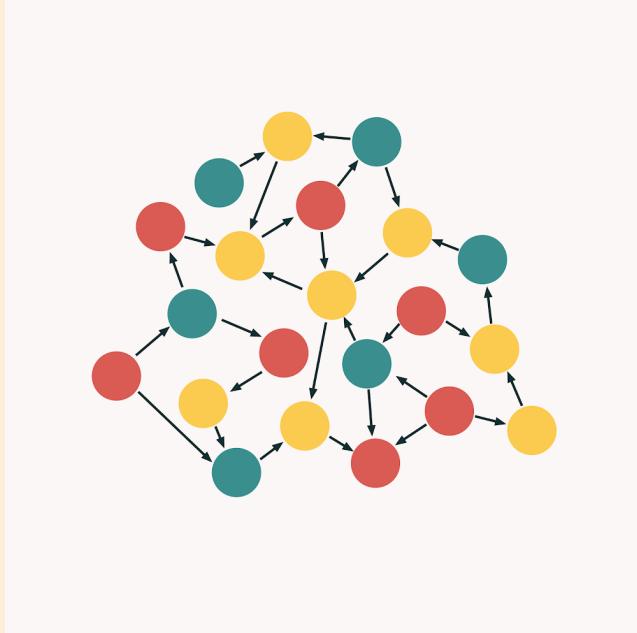


# L'objectif est ...

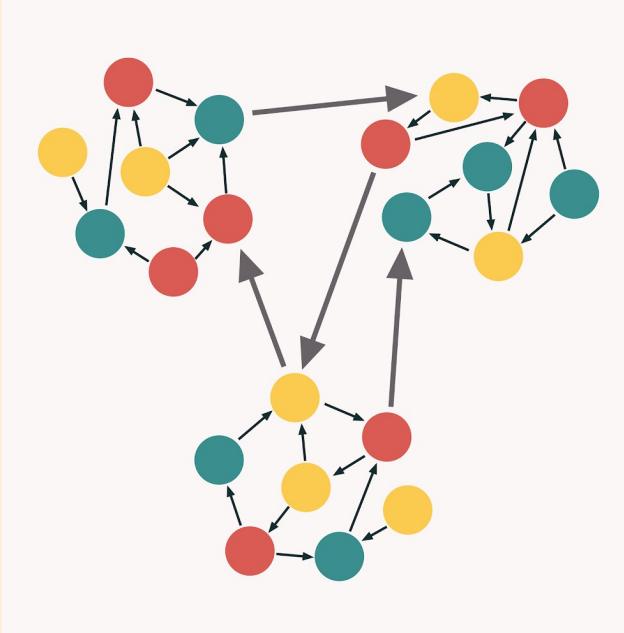
respecter le principe « *Low coupling, High cohésion* » (vue avant)



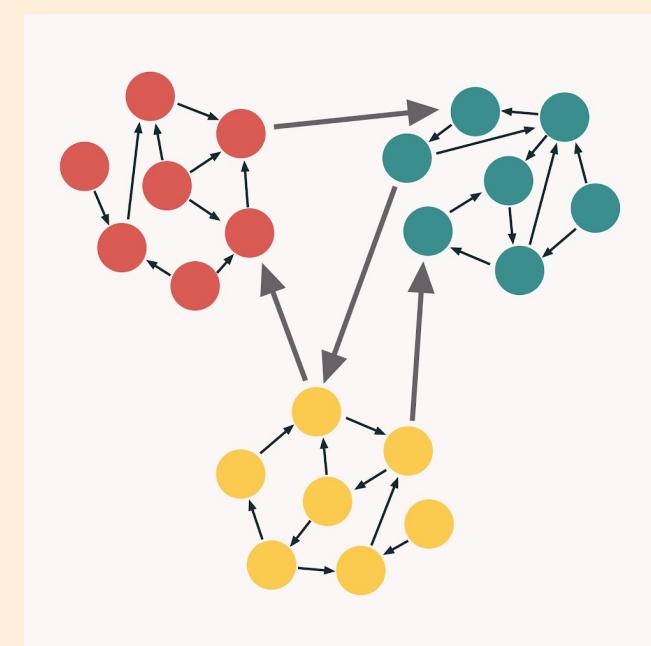
# Illustration



God object



Poorly selected boundaries



Ideal

# Modules et Services

Service

Module A

Module B

Module C

Module n

Rappel : chaque module fournit une interface



# Conclusion de la section

# Conclusion concepts fondamentaux

- Ces concepts sont utilisés partout, tout le temps
- Nous avons vu
  - **Les interfaces** : définissent comment on appelle un service (paramètres) et qu'est-ce qu'on obtient d'un service (type de retour)
  - **L'inversion de dépendance** : éviter de tout package -> build -> test -> deploy
    - (Utilise la notion d'interface)
  - **Qu'est-ce qu'un service** : granularité différente suivant l'architecture
    - (Utilise la notion d'inversion de dépendance)

# Styles architecturaux

3

# Plan concept fondamentaux

1. Introduction générale
2. Architecture monolithique (avantages et inconvénients)
3. Architecture distribuée (avantages et inconvénients)
  
4. Architecture monolithique
  1. Layered
  2. Microkernel
  3. Pipeline
  
5. Architecture distribué
  1. SOA (Service Oriented)
  2. Service Based
  3. Microservices
  4. Event-driven
  
6. Comparaison d'architectures distribuées

# C'est quoi un style architectural ?

Le style Architectural fait référence à un ensemble de lignes directrices ou de principes pour la conception et la construction de systèmes logiciels.

Les Styles Architecturaux (*Architectural Styles*) est le niveau de granularité le plus élevé. Il spécifie les couches, les modules et leurs interactions.

# C'est quoi un style architectural ?

Catégorie	Style Architectural
Communication	SOA, ROA, Message Bus
Déploiement	Client/Server
Domaine	Domain Driven Design, Monolithique, Distribué
Structure	Component-Based, Object-Oriented, Layered, Plug-ins
Autres	REST, Peer-to-peer, Cloud computing, Internet of Things, Blockchain...

*Quelques exemples*

# C'est que n'est pas style architectural ?

Il ne répond pas à un problème précis comme un « patron architectural »

E.g. MVC est un patron architectural qui répond à un problème de séparation UI/Model

MVC est un patron qui s'appuie sur le style Layered

Layered en tant que style ne permet pas de ressourdre un problème précis, c'est juste une architecture récurrente en programmation. C'est juste une façon d'organiser notre code

# Deux sous-ensembles

Styles architecturaux

Architectures Monolithique

Layered

Microkernel

Pipeline

...

Architectures Distribuées

Service Oriented (SOA)

Service-based

Microservices

...

# Architecture Monolithique

# Avantages globaux

- Facilité de développement : tout à un seul endroit
- Facilement déployable : une seule unité
- Peu couteuse (par rapport aux architecture distribuées)

# Inconvénients globaux

- **Déployabilité (faible)** : L'ensemble de l'application ou du site web doit être déployé en une seule unité => Même le plus simple des changements nécessite une construction et un déploiement complets de l'ensemble de l'application.
- **Tolérance aux pannes (faible)** : Du à son caractère monolithique si une partie une de l'architecture provoque une erreur hors mémoire, l'ensemble de l'unité d'application est affectée et tombe en panne.
- **Scalabilité (faible)** : peu de choses peuvent être faites après le déploiement pour réagir à une augmentation de la charge.
- **Élasticité (faible)** : comme pour la scalabilité, ce style d'architecture ne peut pas répondre aux pics d'utilisation au-delà de sa capacité intégrée.

# Architecture Distribuée

# Avantages globaux

- **Résilience** : si un service tombe les autres non
- **Scalabilité/Elasticité** : On va pouvoir adapter le nombres de ressource à la charge. Par exemple, Netflix pique d'utilisateur le soir
- **Agilité** :
  - des équipes plus autonomes; on va pouvoir cloisonner les équipes, chacune travaille sur son microservice => on évite le code spaghetti
  - Et les équipes n'ont pas besoin d'utiliser le même langage ni les mêmes techniques, *on expose juste nos services via interface*

# Inconvénients globaux

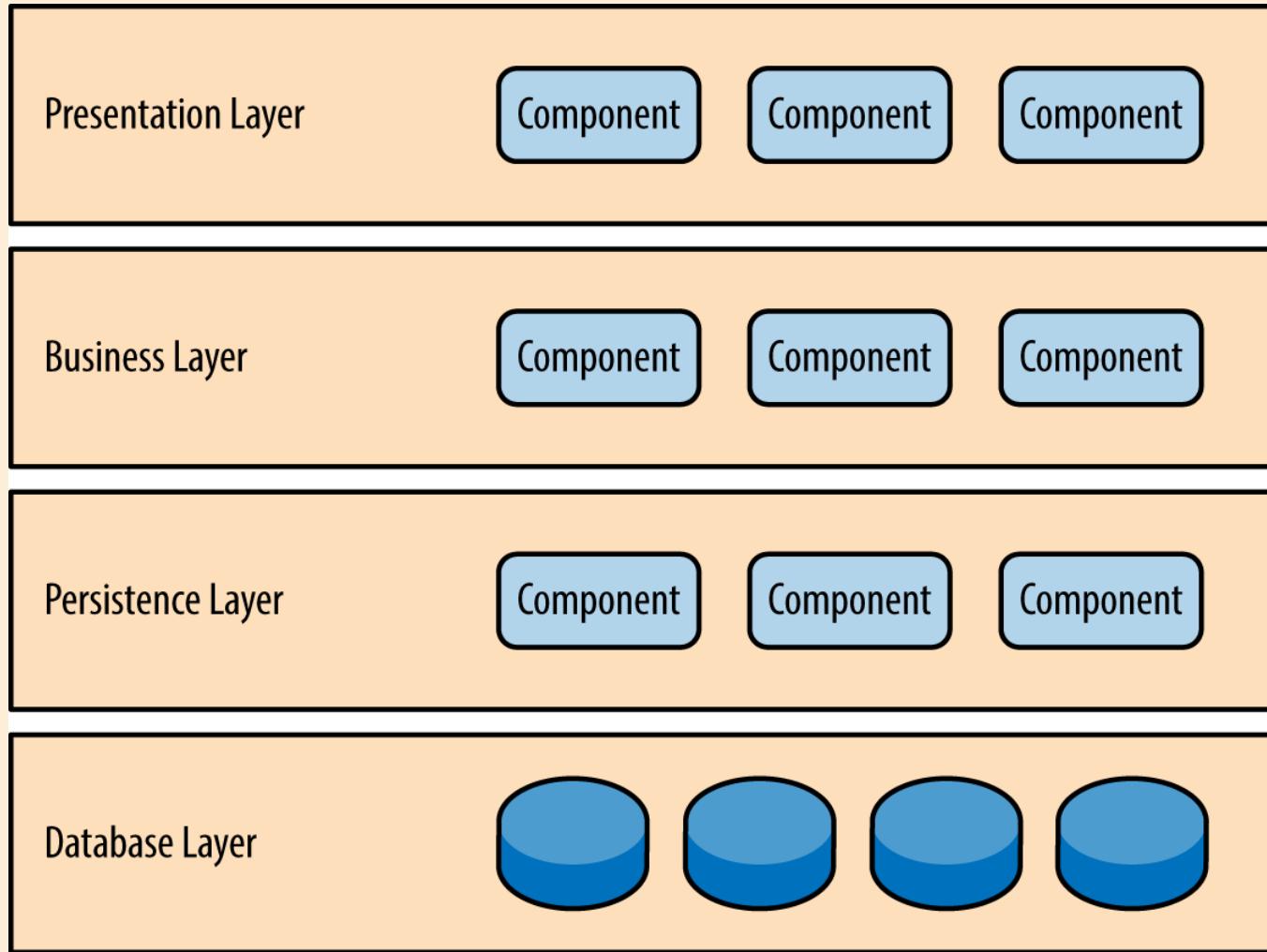
- Compliqué à mettre en place : nécessite des compétences complète (logiciel, réseau, management, etc ...)
- Il faut une équipe dédié pour maintenir l'architecture et le réseaux
- Le cout est très élevé : il faut plus de machines physiques, il faut plus de personnel expert (e.g. nécessite des devops)
  - E.g. Prime Video sont passés du microservice au monolithe pour réduire leur cout de 90.
  - Si 100 microservices => alors 200 réplicas (2 par ms) à maintenir ... => un monolithique coute moins cher

# Layered architecture

(Monolithique)

# Composition

Peuvent être combinées

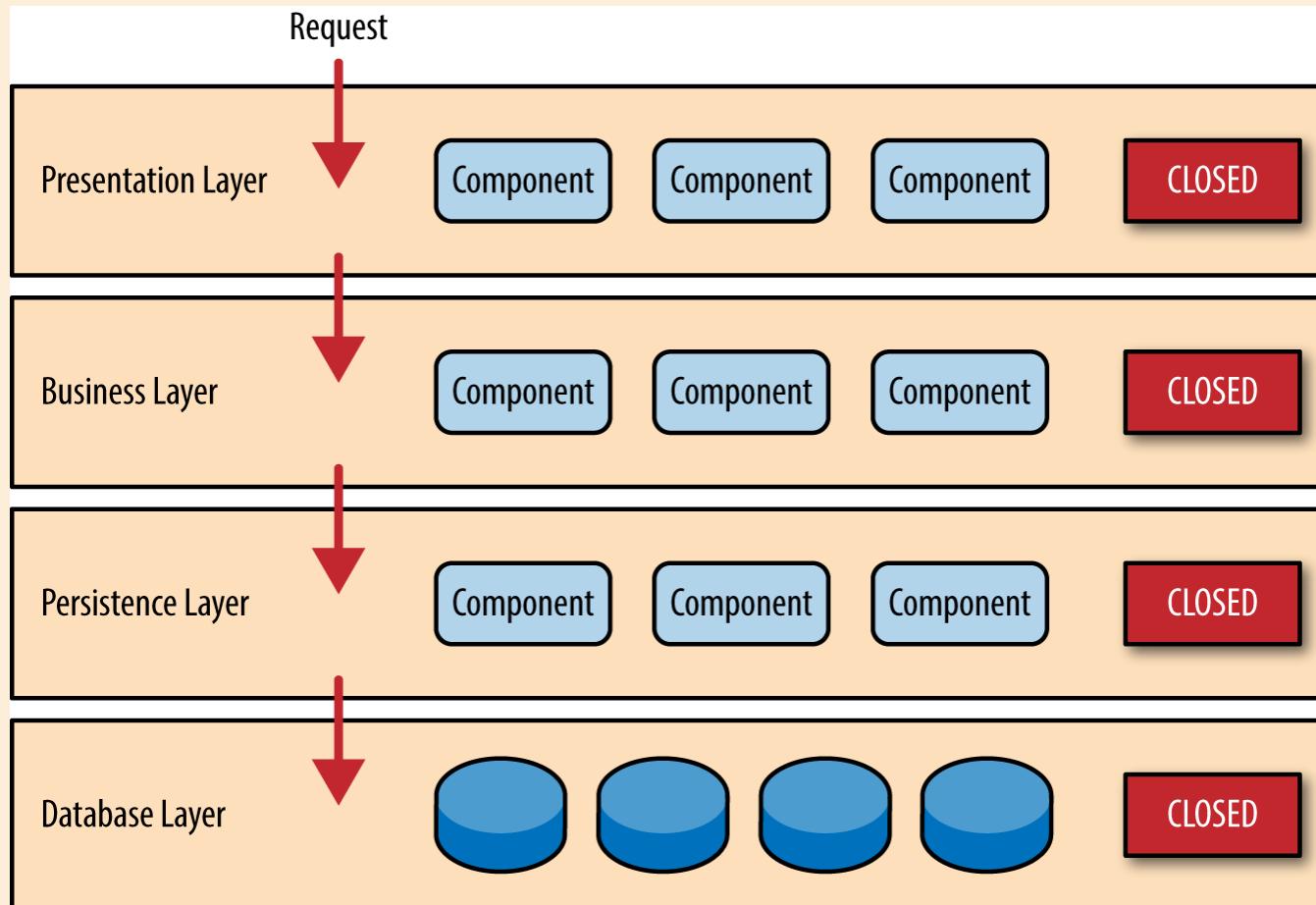


Chaque couche a un rôle bien défini et ne connaît pas les rôles des autres couches

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

***separation of concerns***

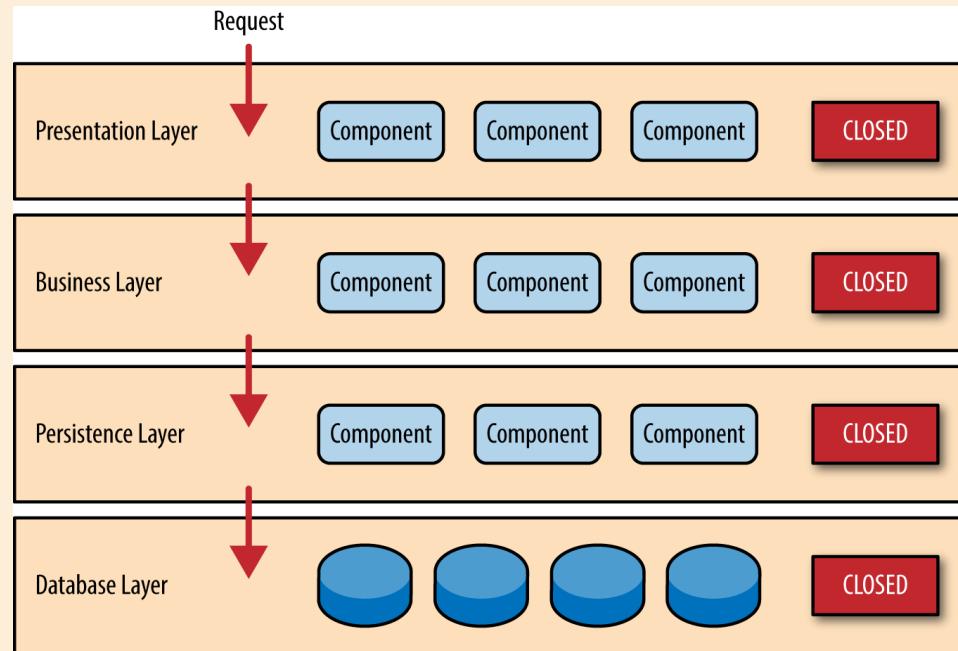
# Couche « closed »



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 ?



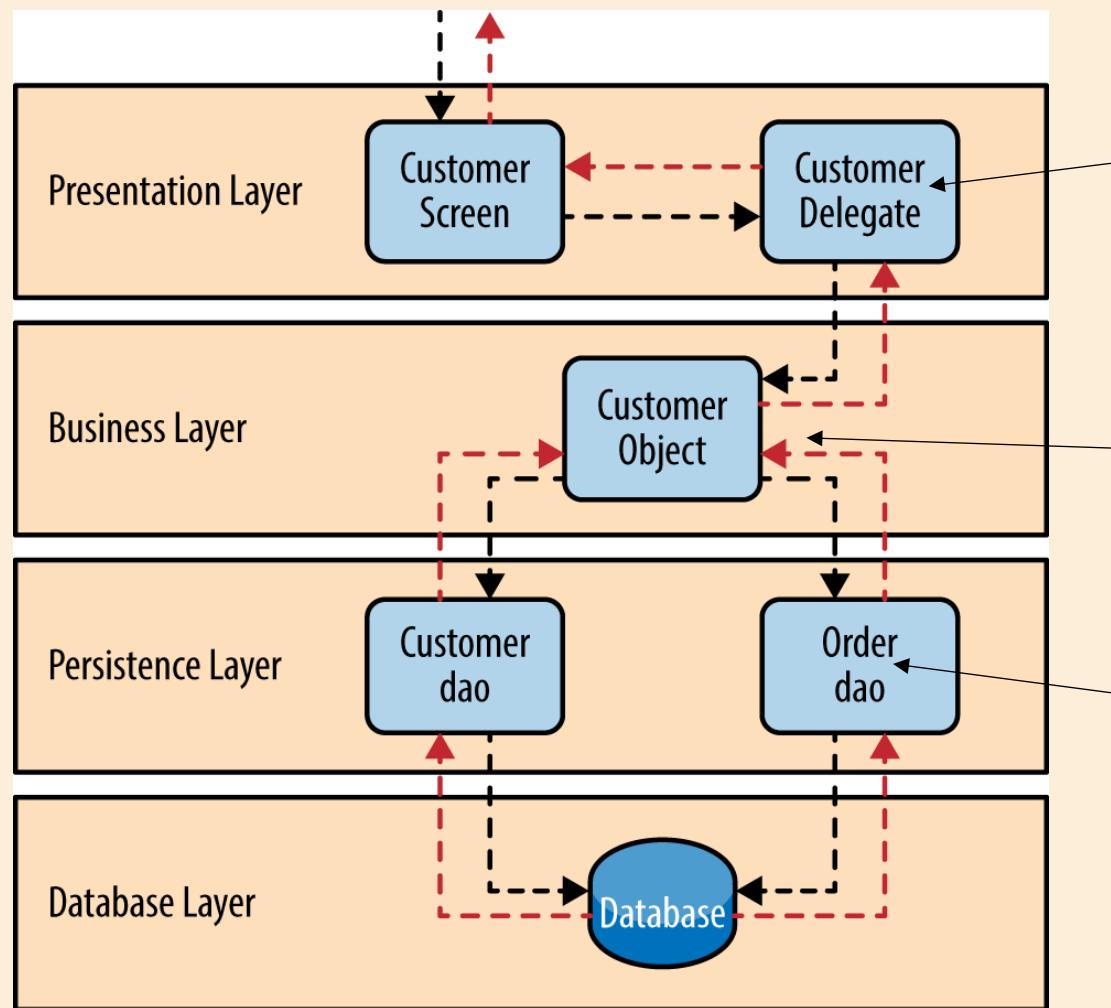
*Il est bien plus rapide de passer de l'IHM à la BDD, les couches ralentissent la requête ?*

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

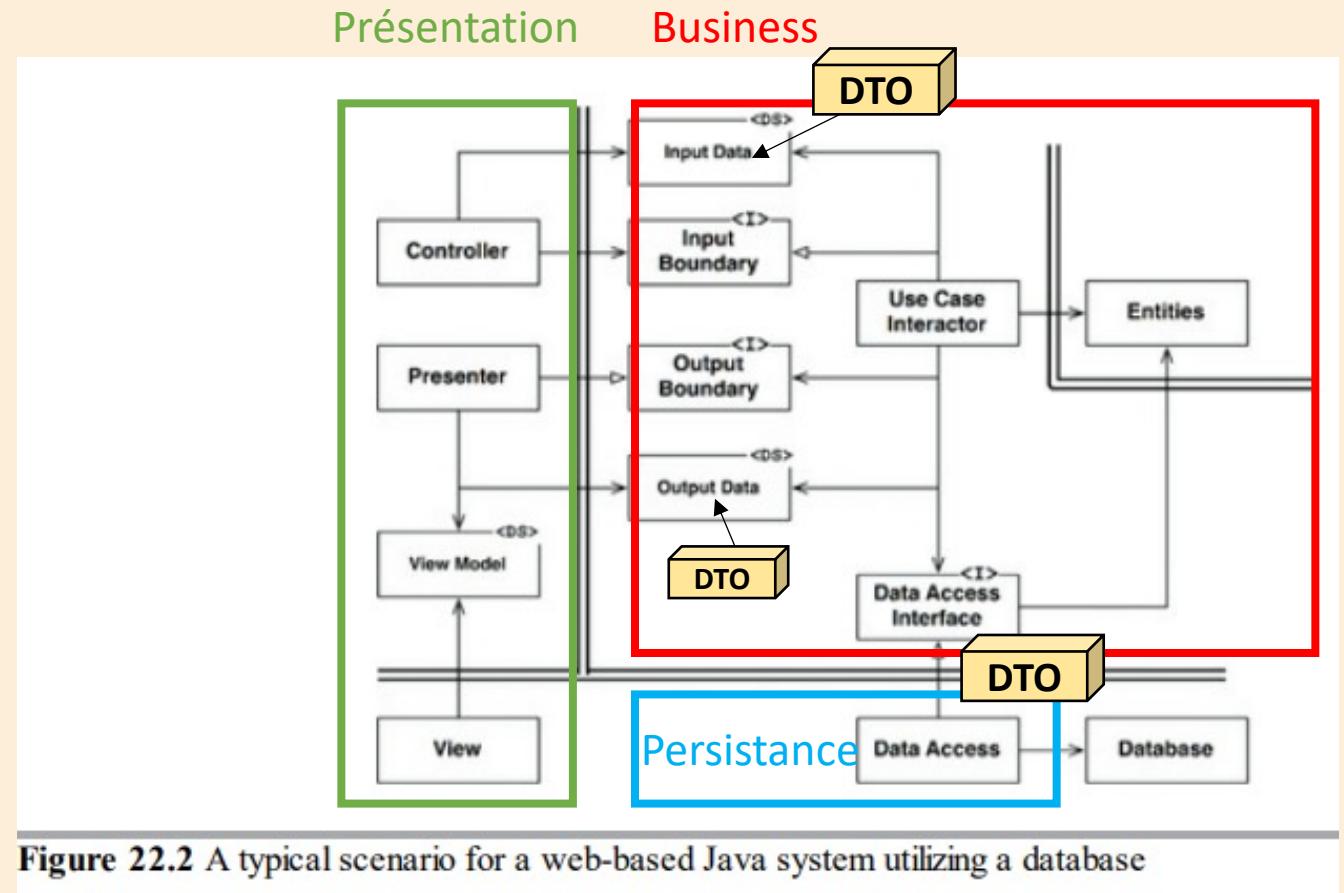
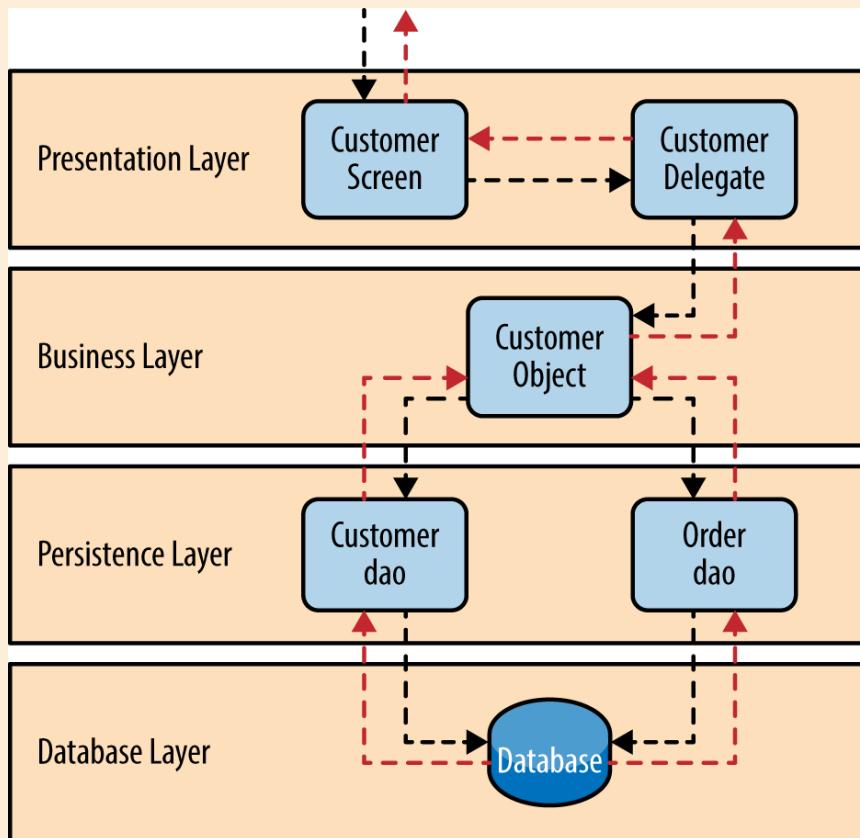


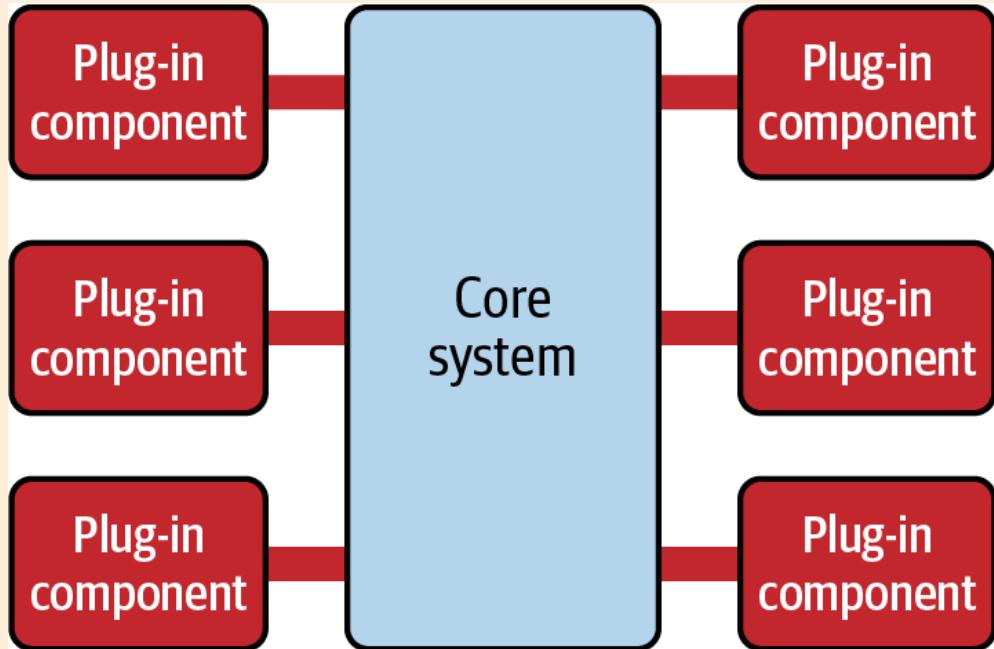
Figure 22.2 A typical scenario for a web-based Java system utilizing a database

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

# Microkernel architecture

(Monolithique)

# 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

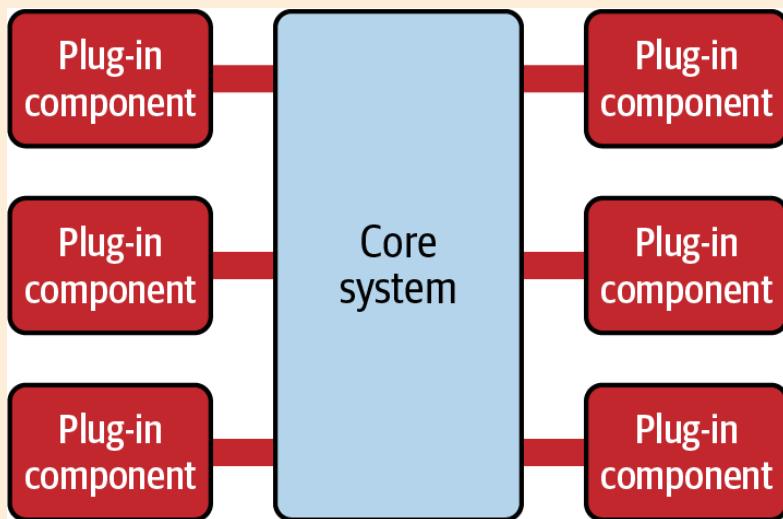
Kernel :

- 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

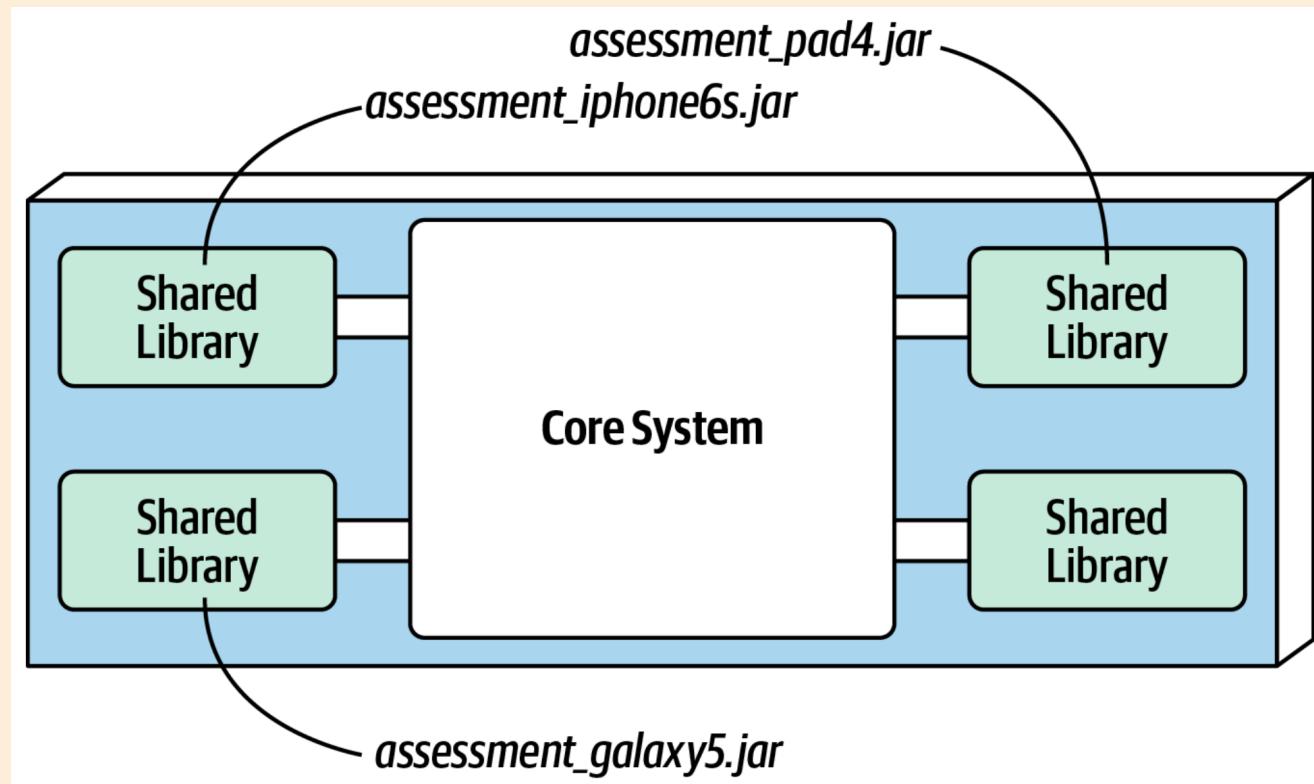


Question :

Quelles solutions pour la communication ?

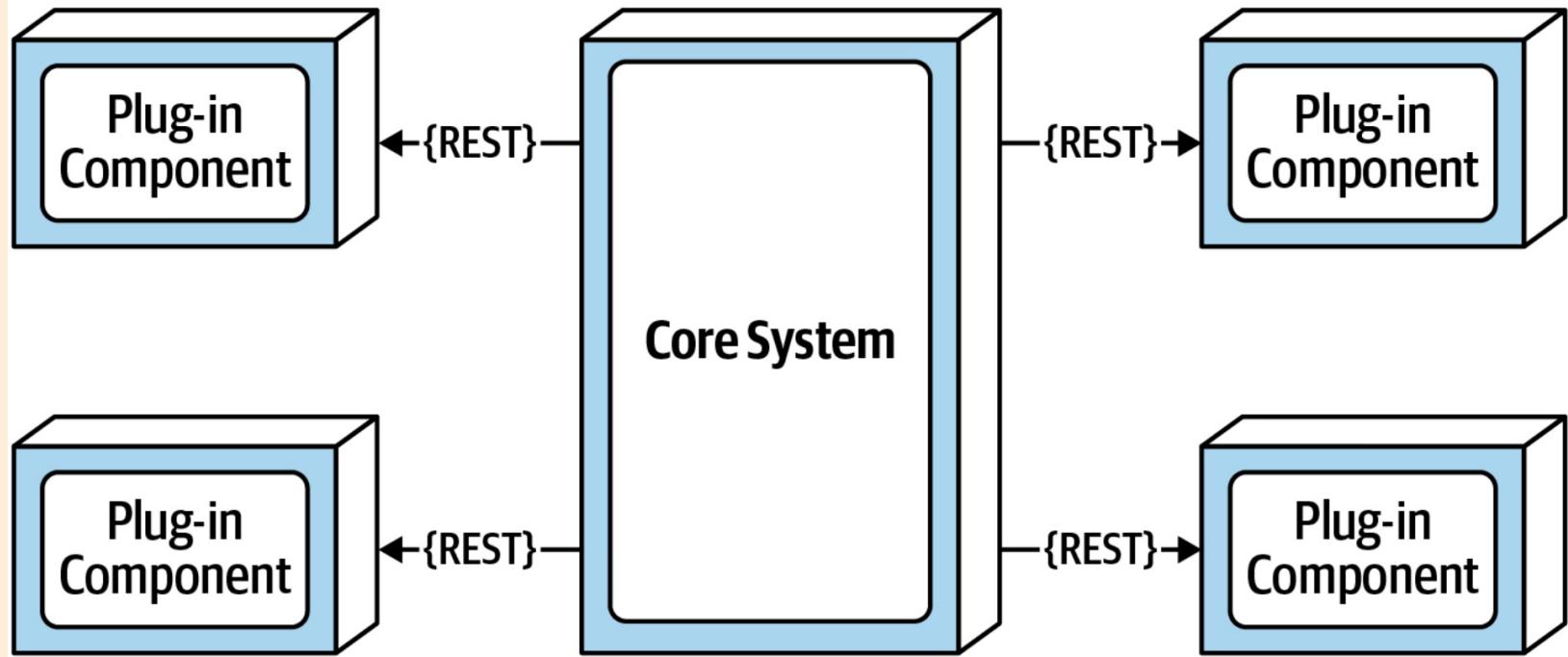
- Point-à-point
- REST  
(explications, slides suivantes)

# 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

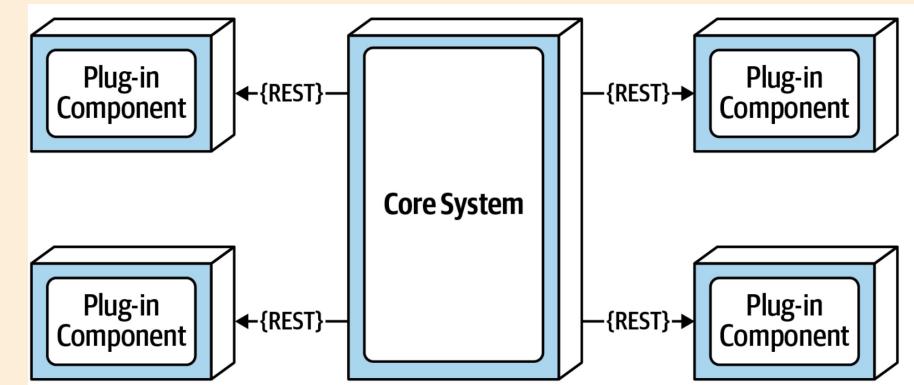


Les modules peuvent également être mis en œuvre comme des services à distance et accessibles par le biais d'interfaces REST à partir du système central.

# Communication : REST

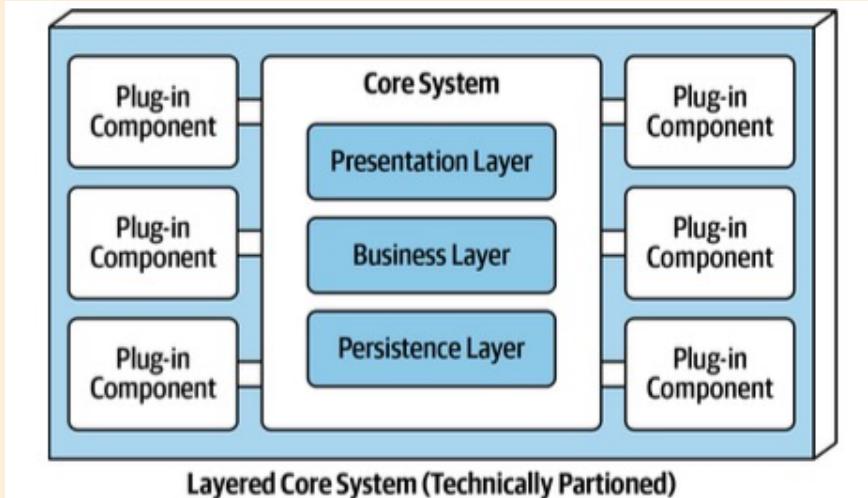
Question :

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

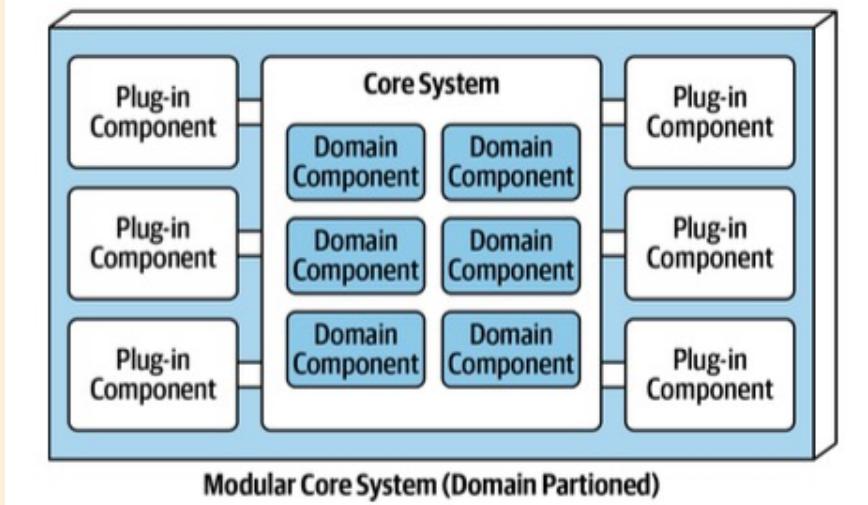


Plusieurs approches existent :

Question :

Comment le Core est implémenté ?

Utiliser une approche par couche



Question :

Comment le Core est implémenté ?

Utiliser une approche par domaine métier

# Pipeline architecture

(Monolithique)

# Composition

Se compose de filtres et de pipes successifs



# Pipe

Les *pipe* servent de conduits pour le flux de données entre les filtres. Chaque *pipe* est généralement unidirectionnelles et la connexion est en point-à-point

# Filtre

Les filtres sont des composants **self-contained**, **indépendants** les uns des autres et **stateless**

Question :  
Stateless ?

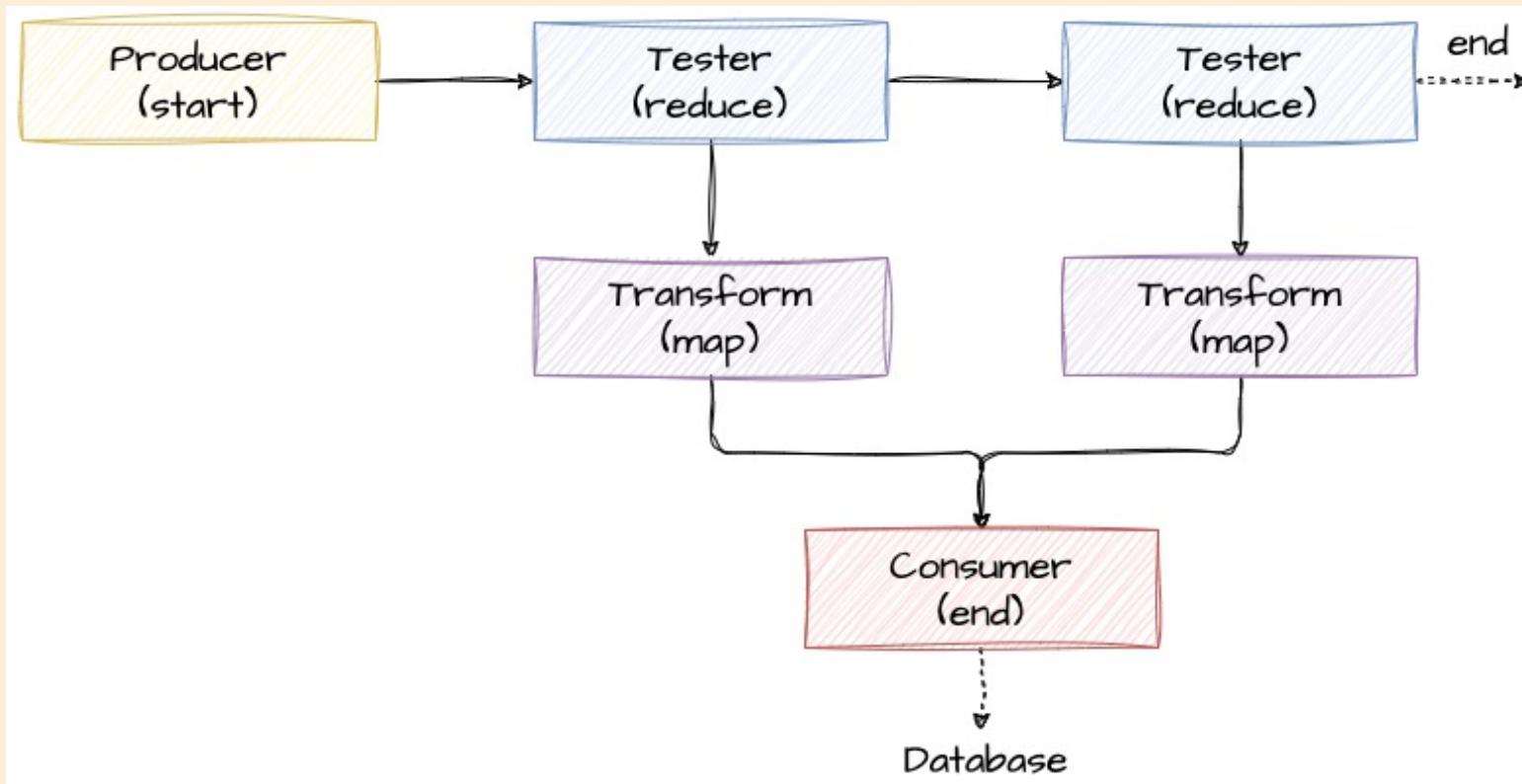
Sans-état

Stateless signifie que le composant ne stocke pas de données et ne fait référence à aucune transaction passée.

Les filtres sont des composants de traitement individuels chargés d'effectuer des tâches spécifiques sur les données.

- Chaque filtre possède une interface d'entrée et de sortie bien définie.
- Les filtres sont conçus pour être modulaires et réutilisables, ce qui facilite l'ajout, la modification ou la suppression d'étapes dans le traitement sans affecter l'ensemble du système.

# 4 types de filtres



**Producer :**

- n'est que sortant, ne reçoit aucune donnée en entrée

**Transformer :**

- Fonction map()

**Tester :**

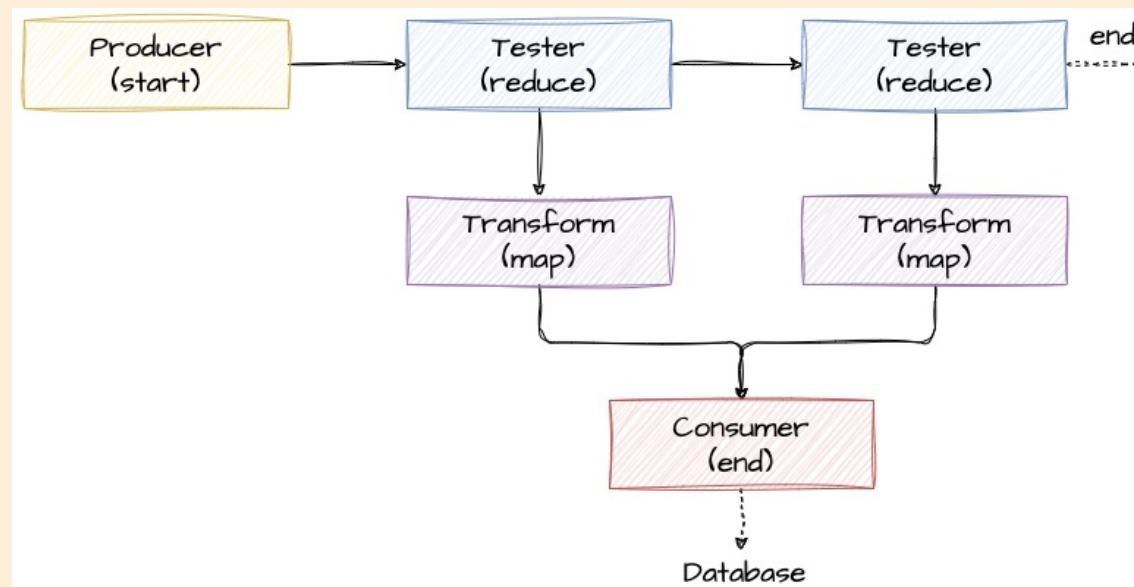
- Ne garde que les données réussissant le test

**Consumer :**

- Persiste en BDD
- Affiche à l'écran

# Avantages

- L'ajout ou la suppression d'un filtre n'impactera pas le système. Par exemple, on peut facilement rajouter un filtre le test
- Des filtres peuvent s'exécuter en parallèle
- On peut avoir plusieurs sorties dans l'architecture



# Service-Oriented Architecture (SOA)

(Distribuée)

# Plan pour l'architecture SOA

- Les concepts présents dans l'architecture SOA sont importants à comprendre car les autres architectures sont des évolutions

## Plan de cours :

- Histoire
- Définition
- Composition

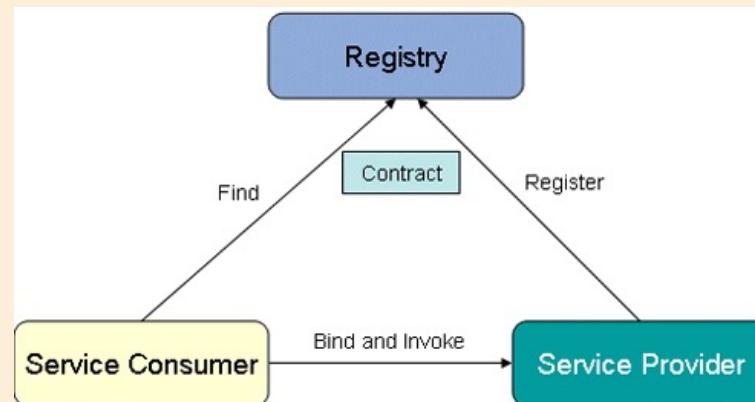
# Histoire

# Histoire de la Service-Oriented Archi.

- Apparu dans les années 1990
  - Concurrence croissante
  - Il faut devenir plus compétitif
  - Il fallait pouvoir intégrer facilement de nouveaux métiers dans les SI des entreprises.
- Pour ce faire, une solution consiste à avoir plusieurs services réutilisables
- ++ Si un service tombe Alors le reste de l'application peut continuer à fonctionner (ce qui n'était pas le cas avec du monolithique)

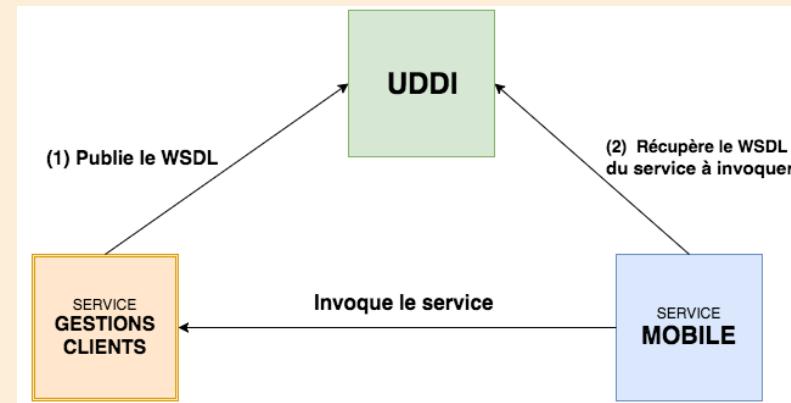
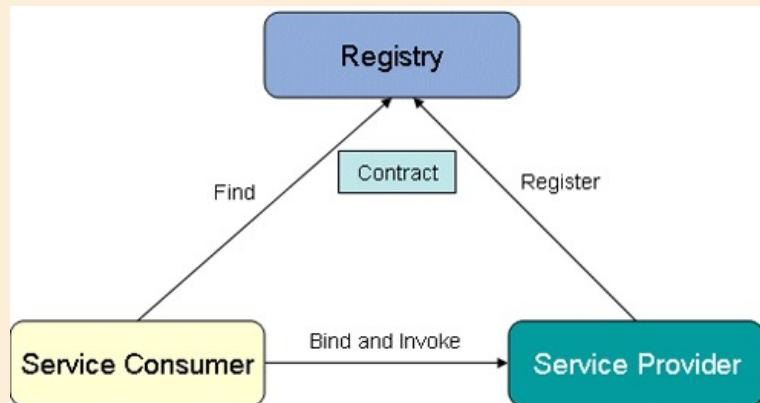
# Caractéristiques SOA

- Une architecture SOA doit répondre au caractéristiques suivantes :
  - n'a pas de langage spécifique de transport et de système d'exploitation.  
Doit être pris en charge par des normes ouvertes.
  - aucun activité humaine n'est impliqué lors de l'exécution
  - application peut être divisée en “fragments de processus d'affaires” (i.g. services). Où chaque service expose une interface bien définie accessible via le réseaux.



# Apparition des Web Service

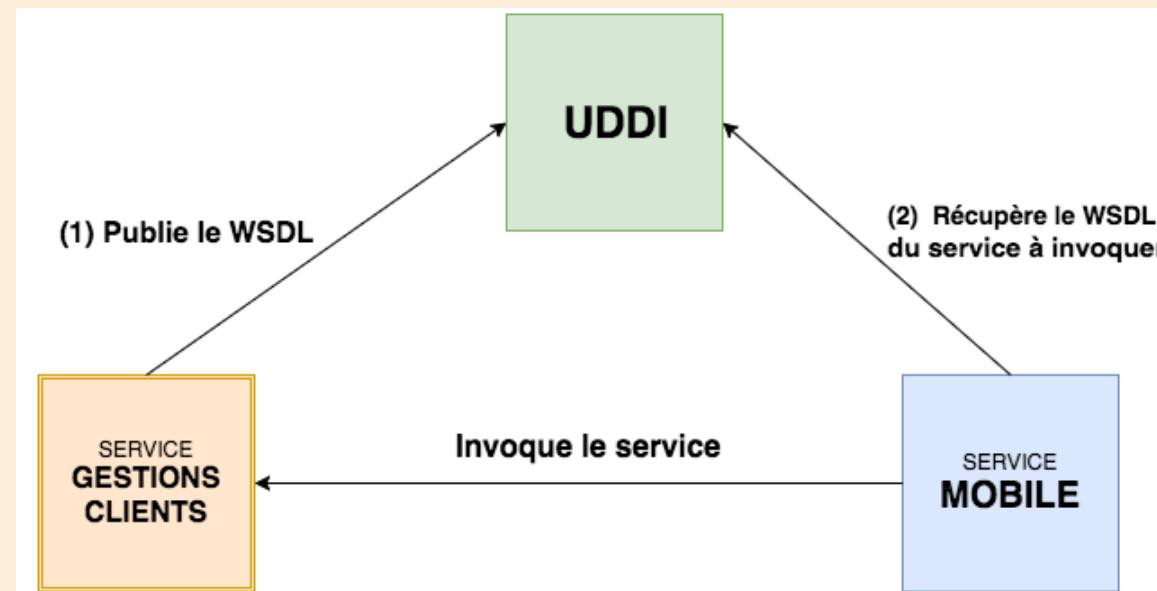
L'architecture SOA et les services web sont deux choses différentes, mais les *services web* sont le moyen privilégié de réaliser l'architecture SOA sur la base de normes.



Explication sur la slide suivante

La figure montre la mise en œuvre de l'architecture SOA avec les services Web :

1. Le fournisseur (service gestion client) publie son contrat d'interface sous la forme d'un document WSDL dans l'UDDI
2. Les demandeurs de services localisent les services à l'aide d'un registre de services, UDDI. Le serveur UDDI contient la base de données des descriptions de services et les fournit à l'application du demandeur de service.
3. Le demandeur (service mobile) utilise le document WSDL pour comprendre le contrat d'interface avec le service web. Grâce aux informations contenues dans le document WSDL le demandeur comprendra comment accéder au service, quelles sont ses méthodes et quels sont les paramètres à envoyer, etc ...



# Définition

# 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.

# Principes de la SOA

- Interopérabilité

Question :

Que veut dire ce terme ?

Tout système client peut exécuter un service, peu importe la plateforme sous-jacente ou le langage de programmation. Par exemple, les processus métier peuvent utiliser des services rédigés en C# et en Python.

Puisqu'il n'y a pas d'interactions directes, les modifications d'un service n'affectent pas les autres composants qui utilisent ce service.

# Principes de la SOA

- Couplage faible

Question :

Que veut dire ce terme ?

- Les services doivent dépendre aussi peu que possible des sources externes
- être sans état (stateless) et ne retenir aucune information des sessions ou des transactions passées
- Ainsi, si vous modifiez un service, cela n'affectera pas de manière significative les applications client et les autres services qui l'utilisent.

# Principes de la SOA

- Service abstraction Principle

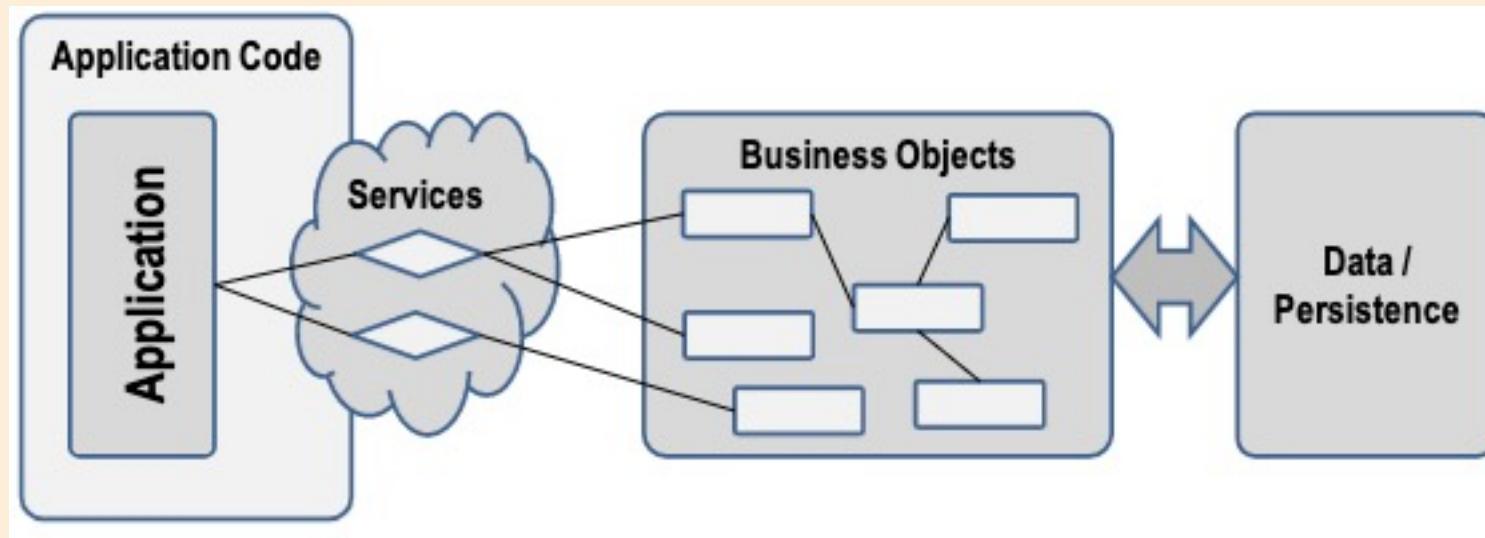
Question :

Que veut dire ce terme ? (indice *abstraction*)

- Les clients ou les utilisateurs de service d'une SOA ne doivent pas connaître la logique de code ou les détails de mise en œuvre du service.
- Ils doivent considérer les services comme une boîte noire.
- Les clients obtiennent les informations nécessaires relatives à la fonction et à l'utilisation du service via des contrats de service et autres documents de description du service.

# Composition

# Composition



Avec la SOA, la logique métier est décomposée :

- en services bien définis et réutilisables
- qui seront exposés pour que tout le monde puisse les utiliser.
- L'application n'a plus qu'à les consommer. Il n'est plus nécessaire de parcourir une hiérarchie d'objets complexe et le développeur n'a plus besoin de comprendre la logique de chaque domaine.

# Composants

Le Service

Fournisseur de  
services

Consommateur  
de services

Registre de  
services

# Composant : service

L'architecture orientée services expose les fonctionnalités de l'entreprise sous la forme de services destinés à être consommés par les applications.

De manière individuelle, chaque service est doté de trois fonctions principales :

- **Mise en œuvre du service** : le code
- **Contrat du service** : prérequis, le coût et la qualité
- **Interface du service** : comment on invoque le service, le retour du service

# Composant : fournisseur de services

Le fournisseur du service crée, maintient et fournit un ou plusieurs services que d'autres entité peuvent utiliser.

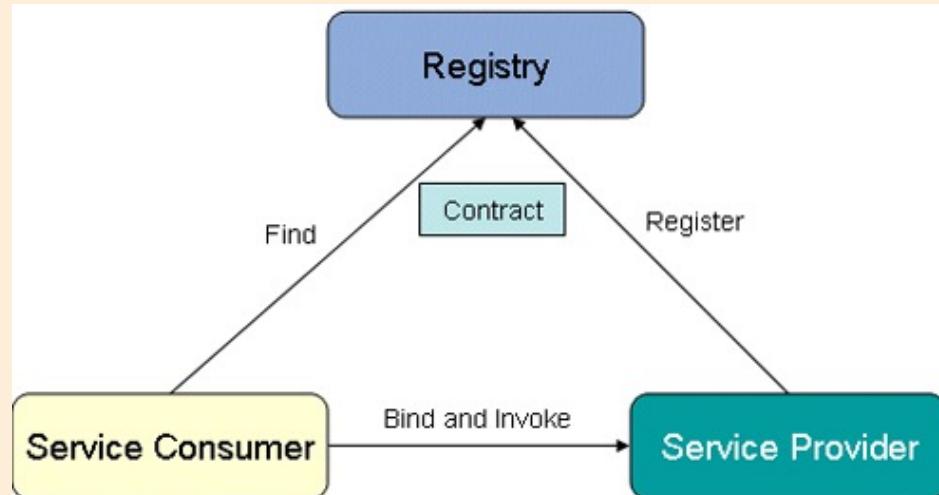
# Composant : consommateur de services

Le consommateur de services demande au fournisseur de services d'exécuter un action spécifique.

Le contrat de service (accessible via le registre) spécifie les règles que le fournisseur et le consommateur de services doivent suivre lorsqu'ils interagissent l'un avec l'autre (e.g. protocole, paramètres d'entrée, paramètre de sortie, etc ...)

# 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.



# Enterprise Service Bus (ESB)

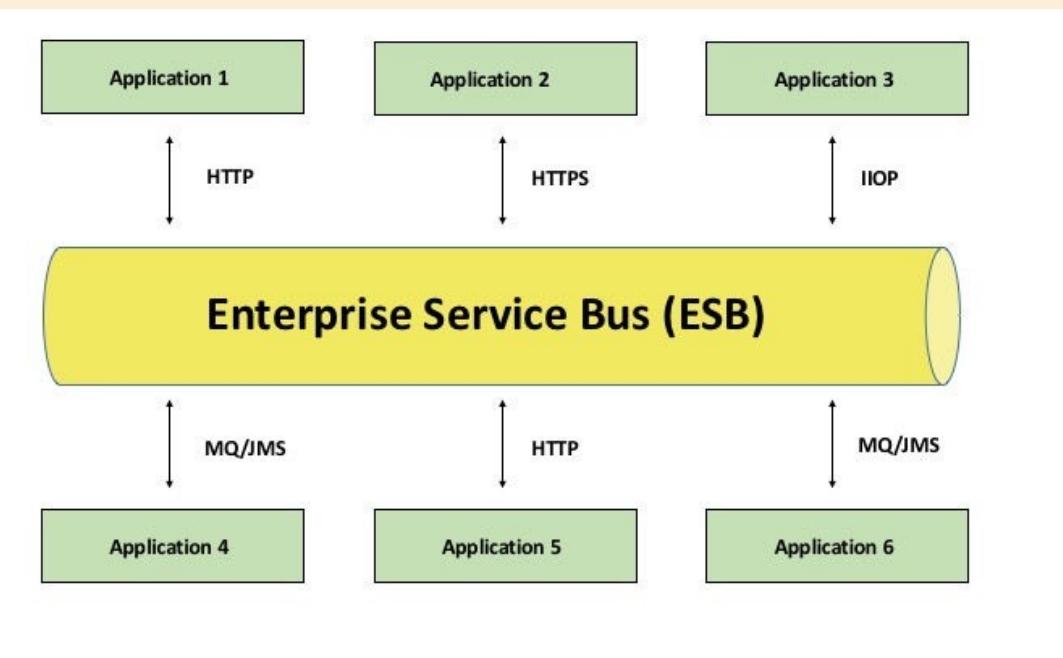
Nous avons vu que notre SI est composé de plusieurs services. Mais ces services n'étant pas forcément écrit dans un même langage, n'utilisant pas les mêmes protocoles doivent pouvoir fonctionner ensemble.

**Question :**

Comment assurer la communication entre nos différents services ?

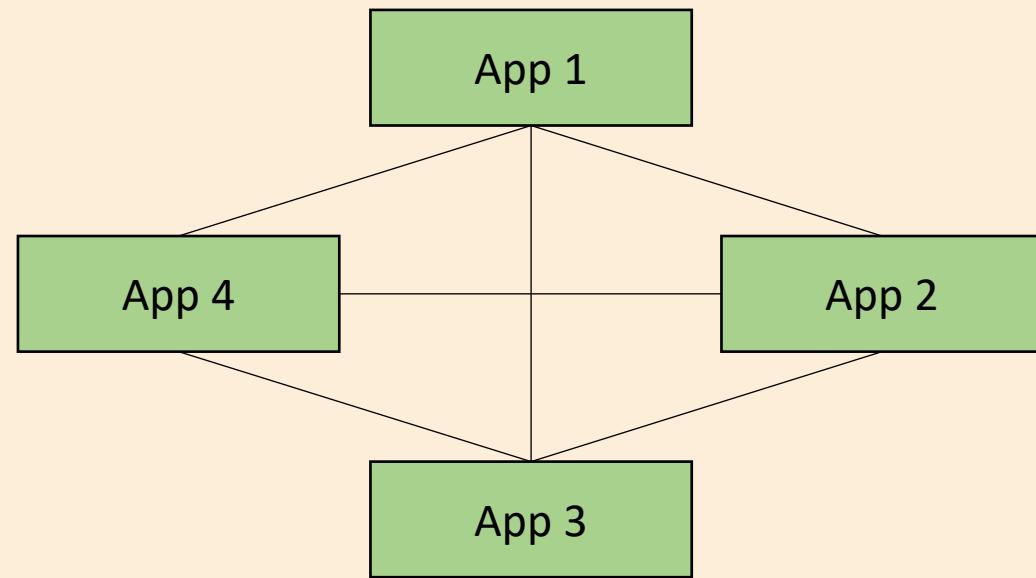
# ESB : définition

Un ESB fournit des capacités de communication et de transformation à travers une interface de service réutilisable. Vous pouvez considérer un ESB comme un service centralisé qui achemine les demandes de service vers le service approprié.



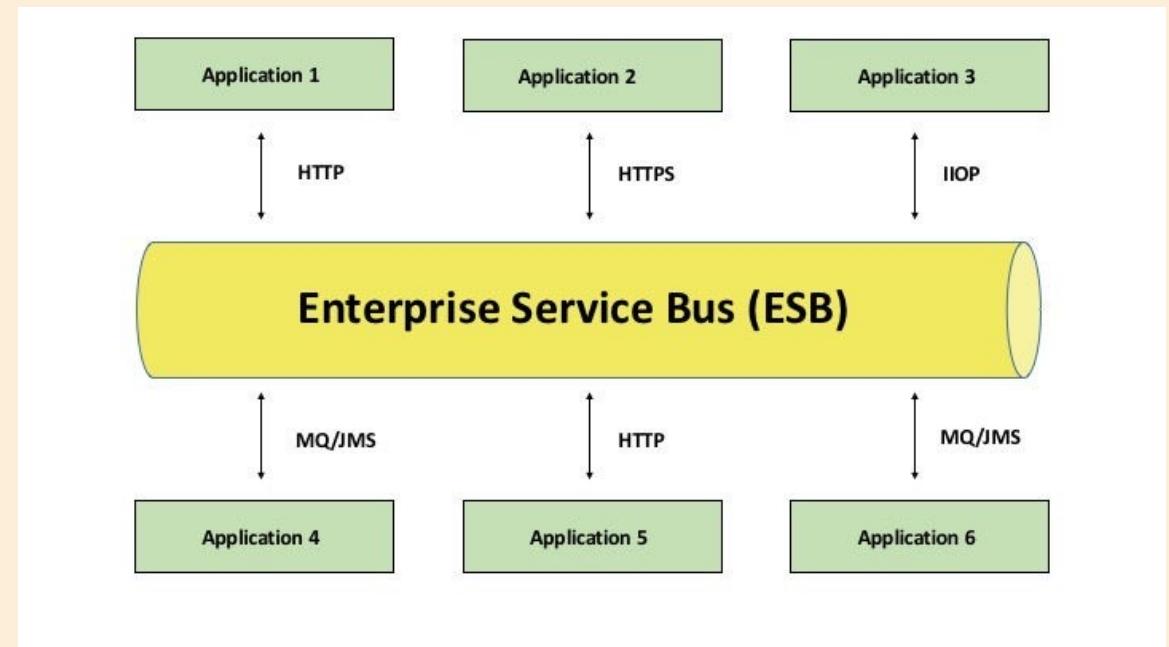
# ESB : Objectifs

HTTP, MQ, ... sont des protocoles ouverts



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

# ESB : Objectifs

L'ESB s'occupera de faire le nécessaire pour transformer votre message et l'adapter avant de le transférer au composant demandé. Il fera la même chose pour la réponse.

# Service-based architecture

(Distribuée)

# Définition par Mark Richards

*Service-based architecture is a hybrid of the microservices architecture style and is considered one of the most pragmatic architecture styles, mostly due to its architectural flexibility. Although service-based architecture is a distributed architecture, it doesn't have the same level of complexity and cost as other distributed architectures, such as microservices or event-driven architecture, making it a very popular choice for many business-related applications.*

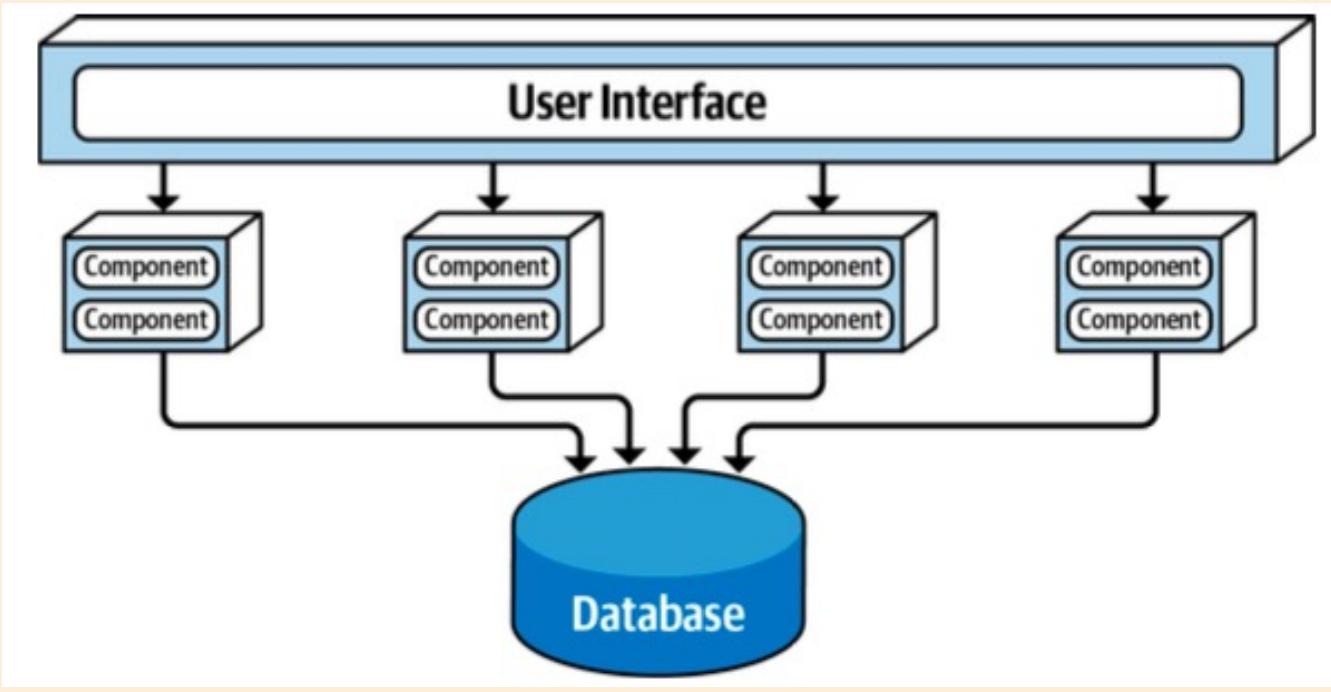
Question :

Que comprenez vous de cette définition ? (réponse slide suivante)

# Pourquoi cette architecture

- Monolithique == 1 gros composant
- Microservices == plusieurs centaines de composants, coût très élevé
- **L'architecture Service-Based est un intermédiaire entre les architectures monolithiques et l'architecture microservices**
  - Elle a entre **4 et 12 services**

# Composition



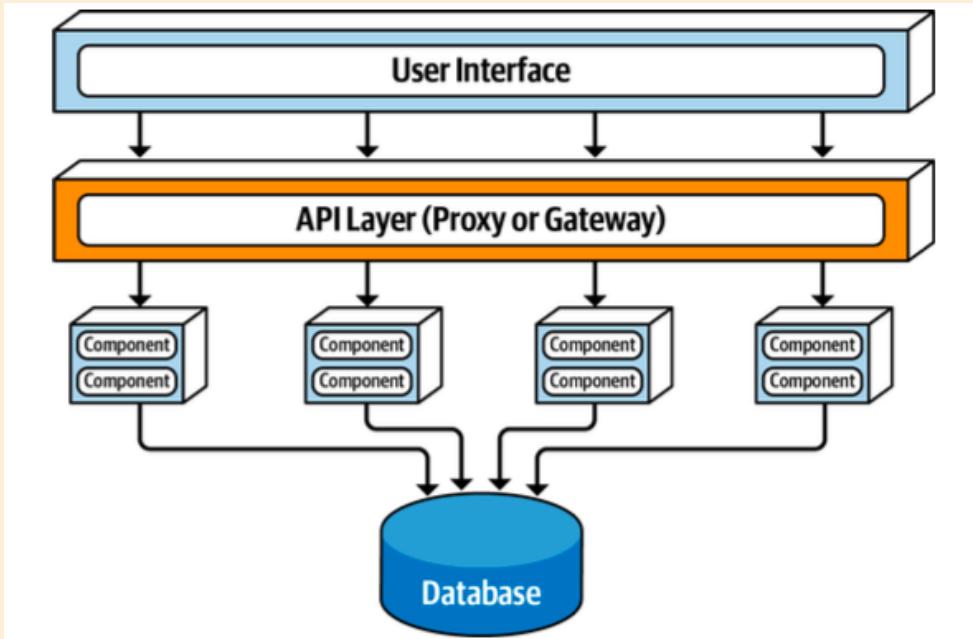
De 4 à 12 services *maximums*

- Plusieurs services (domaines indépendants) indépendant (seule la BDD est une dépendance commune)
- d'une base de données centrale qui est partagée avec l'ensemble des services
- les services sont accessibles via un protocole d'accès à distance (e.g. REST)

Question :

Que pouvez vous me dire de la taille des services ?

# Composition : API Layer



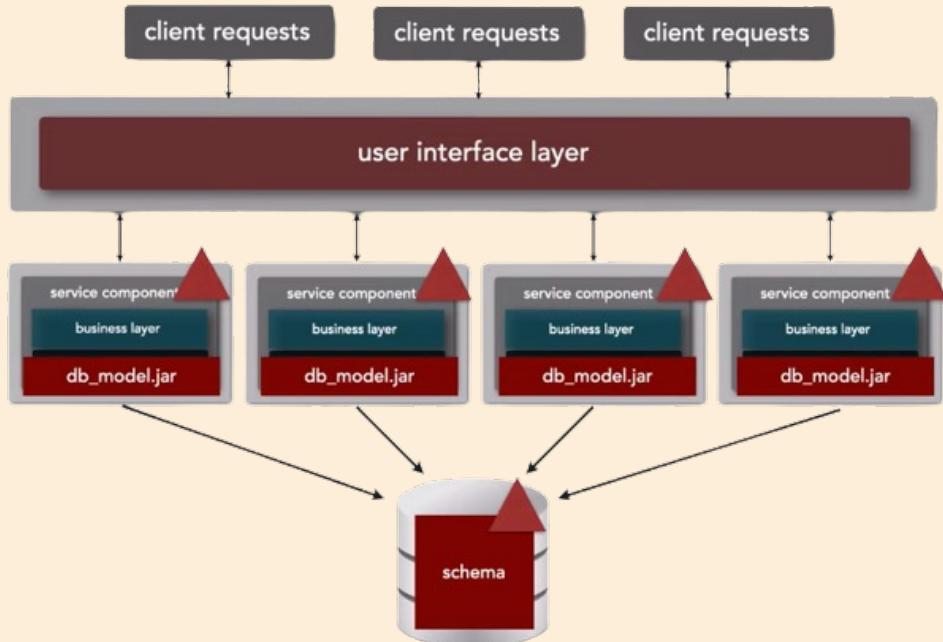
Question :

Pourquoi cette pratique ?

Les systèmes externes communiquent avec nos services uniquement au travers de la couche API.

Cette couche peut par exemple s'assurer que l'utilisateur ait le droit de consulter la ressource (authentification et autorisation).

# Composition : partitionner la BDD



## Question :

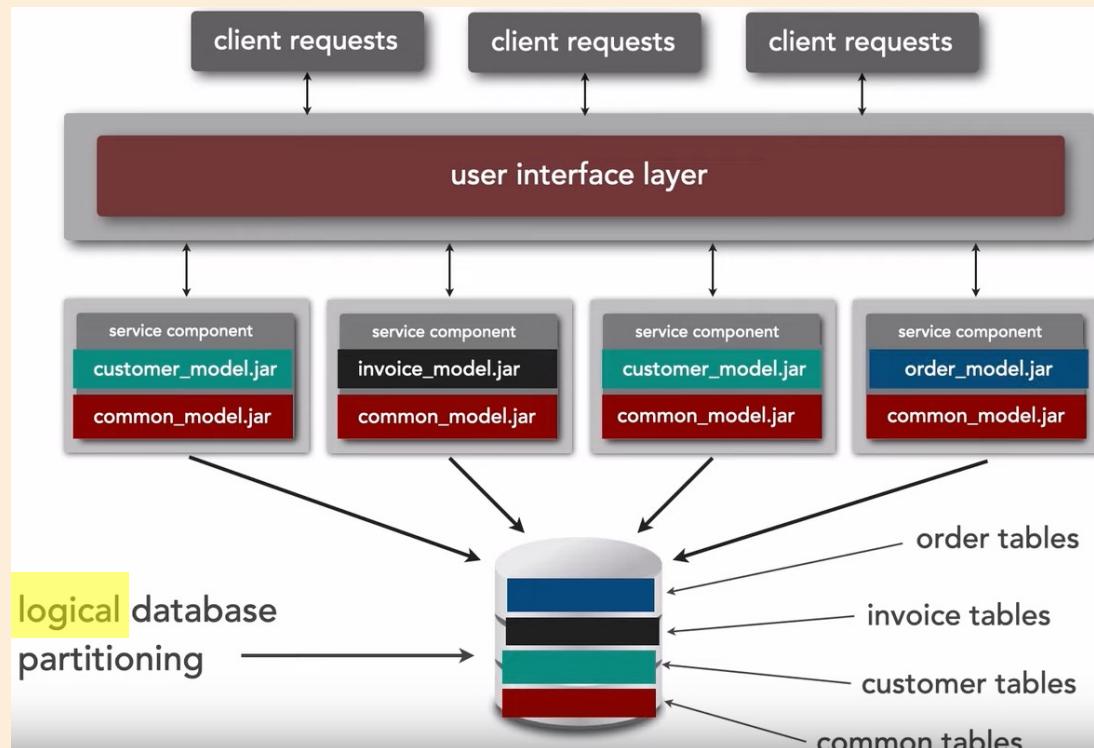
Quel est le problème d'une BDD unique en terme de dépendances ?

Un changement des règles dans le schéma de la base de données entraînera une mise à jour de db\_model.jar donc tous les services se retrouvent impactés. Pour éviter ceci, il faut que chaque service partage son jar.

# Composition : partitionner la BDD

Question :

Quelle solution me proposez vous ?



Une façon de procéder est d'utiliser un *partitionnement logique de la base de donnée* (et non physique). Nous pourrons avoir les partitions suivantes :

- une partition avec toutes les données communes
- des partitions pour les domaines spécifiques

Par conséquent, si on doit mettre à jour le modèle de base de données *commande* seul le service *Commande* sera impacté. Les autres ne partageant aucune données n'auront pas besoin d'être retesté ni redéployé.

# Microservice architecture

(Distribuée)

# Plan pour l'architecture microservices

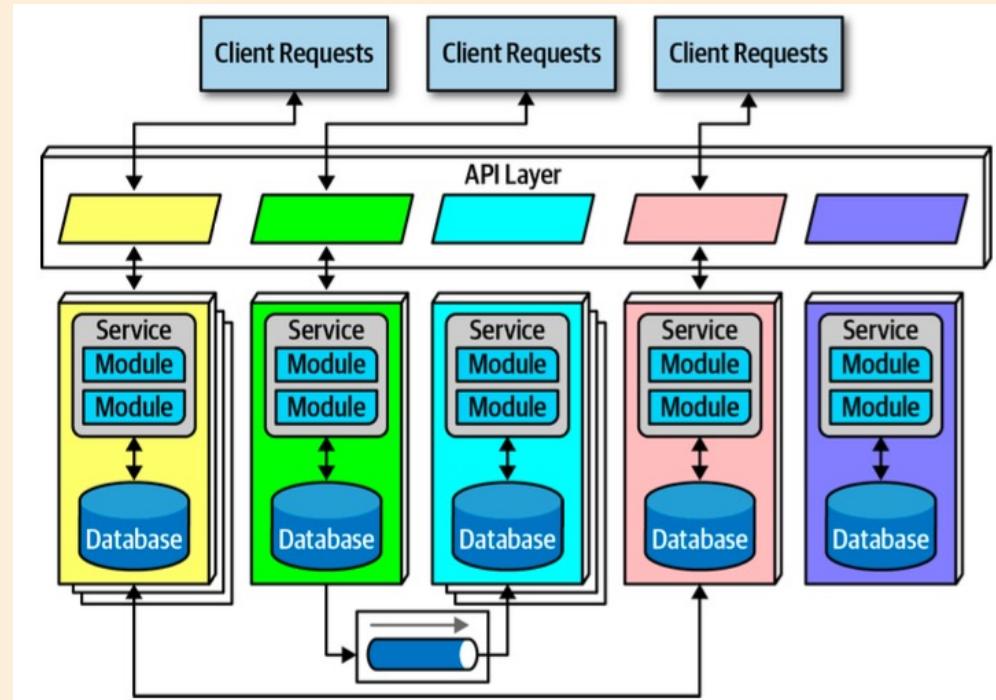
## Plan de cours :

- Brève présentation
- « Bounded Context »
- Front End
- Communication
  - Chorégraphie/Orchestration
  - Transactions

# Brève présentation

# Brève présentation

Elle prend son inspiration du Domain-Driven Design (DDD), une approche conceptuelle et méthodologique du développement logiciel qui met l'accent sur la modélisation du domaine métier. Notamment de la notion centrale de *Bounded Context*.

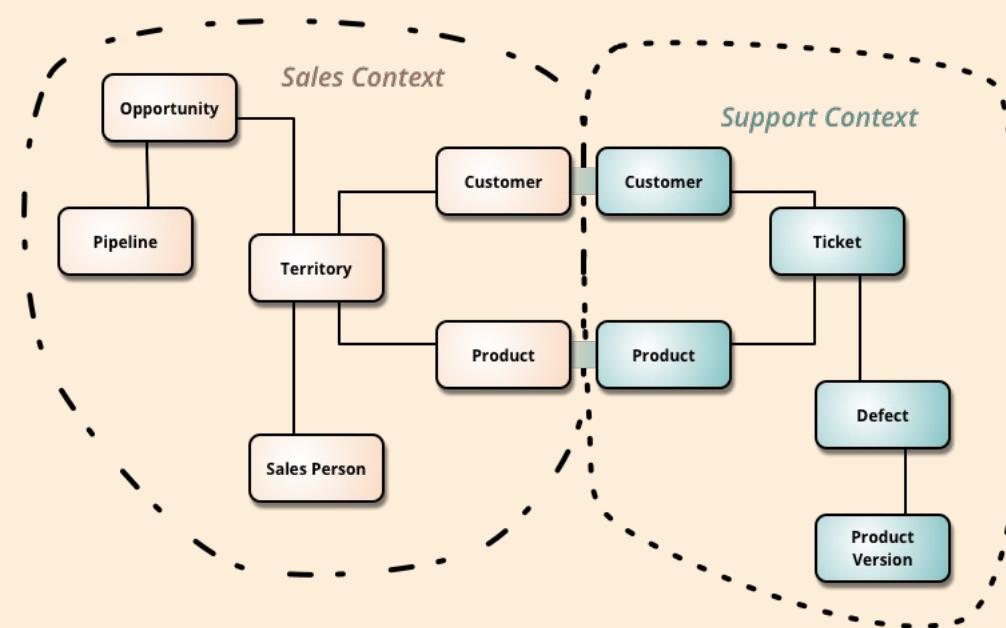


# Bounded Context

# Bounded Context

Chaque service comprend tout ce qui lui est nécessaire pour fonctionner, classes, sous-composants, base de données.

Il s'agit d'un moyen de segmenter un système logiciel plus vaste en parties plus petites et plus faciles à gérer.



# Bounded Context : granularité

Question :

Quelle est la principale question à laquelle nous devons répondre ?

Question :

Donc « micro » dans microservice signifie ... ?

A cause du terme *micro* des personnes commettent souvent l'erreur de rendre leurs services trop petits, ce qui les oblige à établir des liens de communication entre les services pour effectuer un travail utile. (+ voir slide suivante)

# Bounded Context : granularité

Nous devrions réunir ensemble les choses qui ont un lien

Créer du lien c'est créer du couplage, créer du couplage montre une forte dépendante; pourquoi pas les réunir dans un seul et même service ?

# Bounded Context : granularité

Pour trouver les bonne frontière on peut suivre ces recommandations :

- **Objectif** : chaque service doit avoir une très forte cohésion fonctionnelle. Contribuant à un comportement significatif de l'application.
- **Transaction** : les transactions posent des problèmes dans les architectures distribuées si on veut les éviter le mieux est de regrouper les services les partageant.

Question :

Pourquoi ?

# Front end

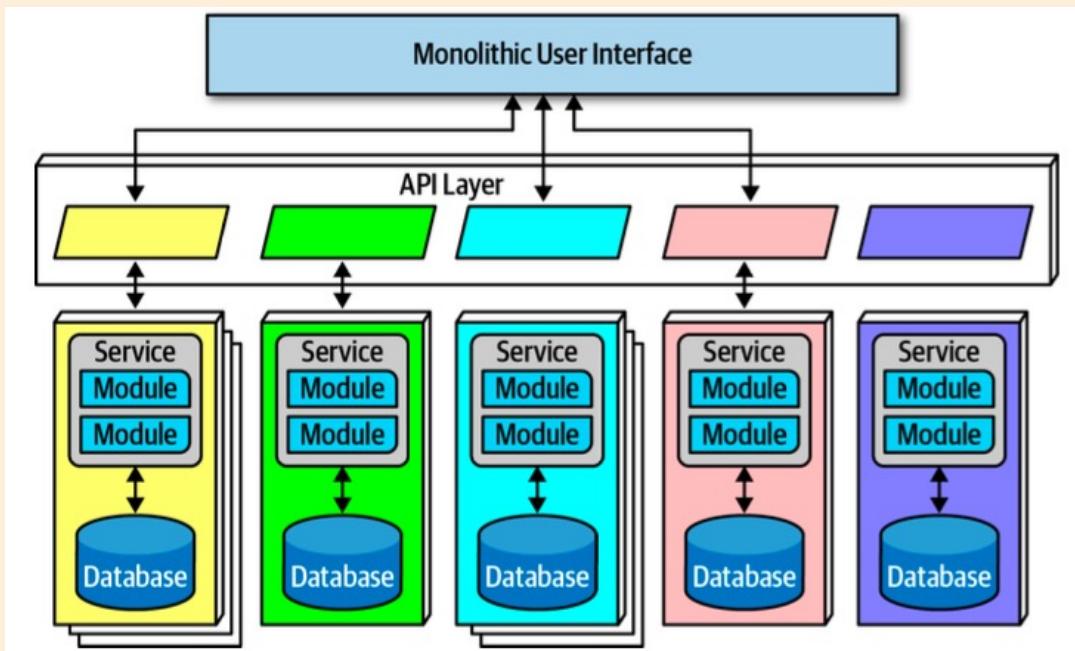
# Front end

Dans la version initiale de l'architecture microservices la couche de présentation était inclus dans les Bounded Context, mais cette approche rendait l'architecture compliquée.

On va donc sortir l'interface graphique (la vue) pour la gérer en dehors de bounded context

# Front end : approche 1

## Monolithique user interface



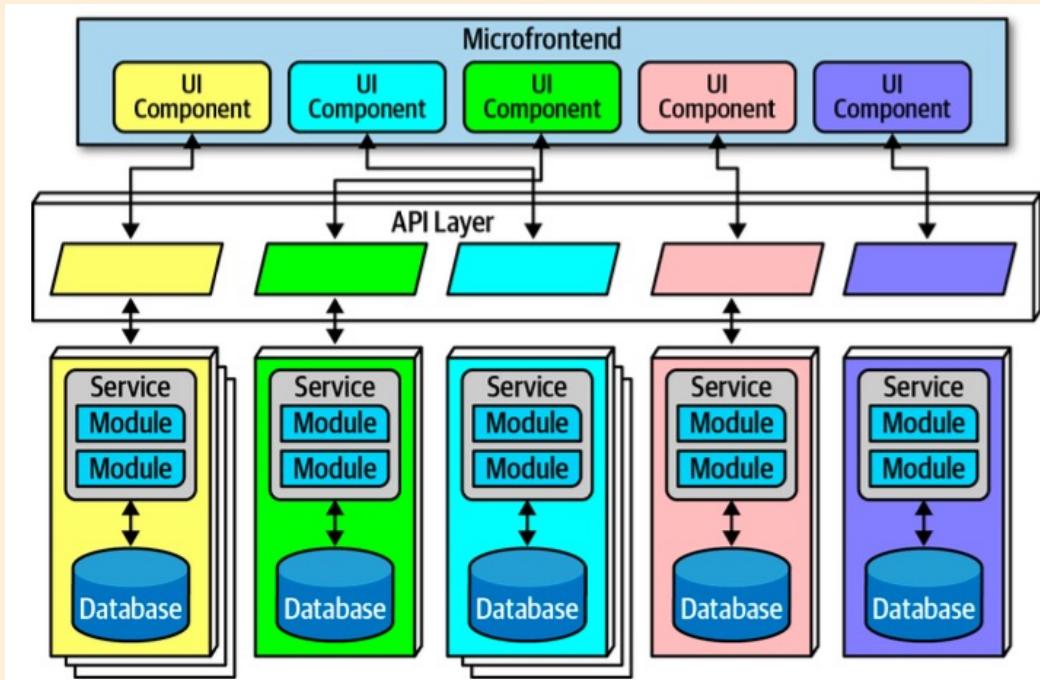
Une interface utilisateur unique qui fait appel à la couche API pour répondre aux demandes des utilisateurs. Il peut s'agir d'une application bureautique, mobile ou web. Par exemple, de nombreuses applications web utilisent désormais un cadre web JavaScript pour construire une interface utilisateur unique.

# Front end : approche 2

Question :

A votre avis ?

## Microfrontend user interface



Question :

Quel est l'avantage pour les équipes de dev ?

En utilisant ce modèle, les équipes peuvent isoler les limites des services depuis l'interface utilisateur jusqu'aux services backend.

Ainsi une équipe de développement peut faire l'ensemble sur service de A (frontend) à Z (backend).

# Communication

Deux notions :

- Chorégraphie/Orchestration
- Les transactions

# Chorégraphie/Orchestration

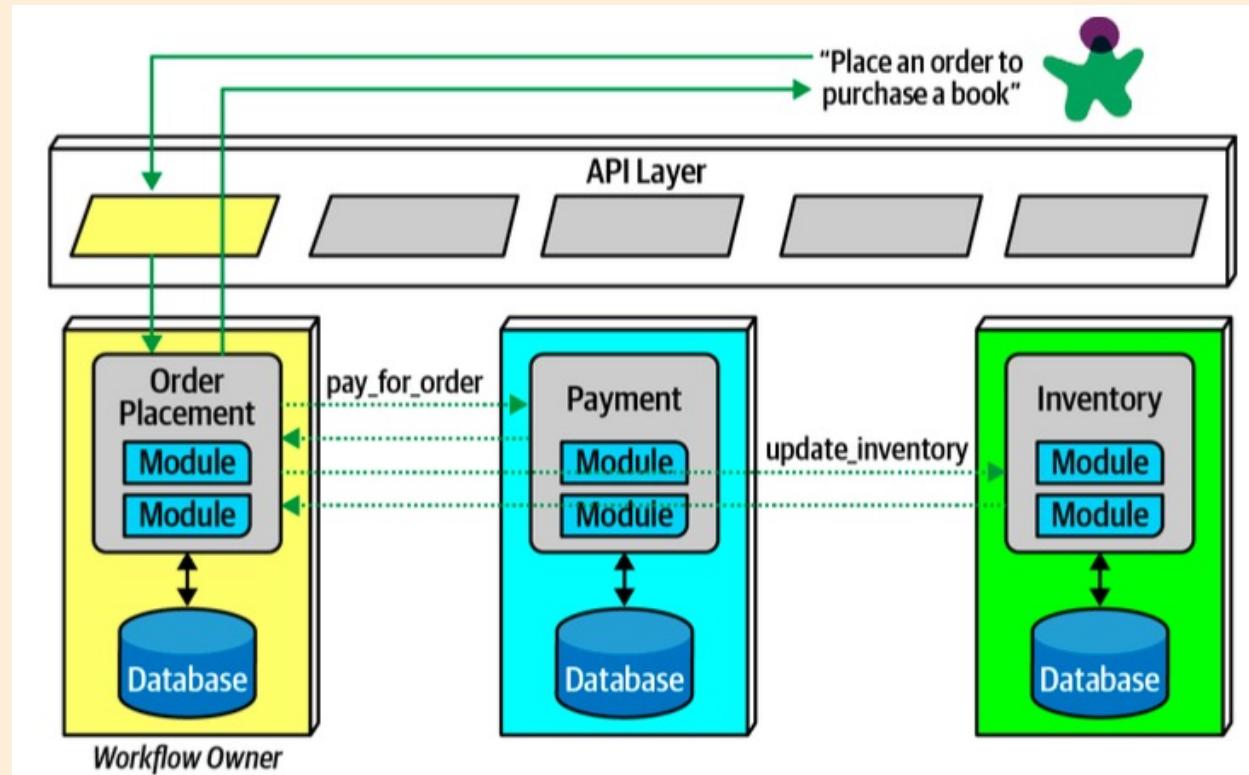
Permet la communication inter-microservice

Les microservices ont parfois besoin d'informations contenu dans d'autres microservices, il faut donc qu'il puisse récupérer ces informations

Dans l'architecture service-based on devait limiter au maximum les communications inter-services. Avec l'architecture microservices, ces interactions sont monnaies courantes

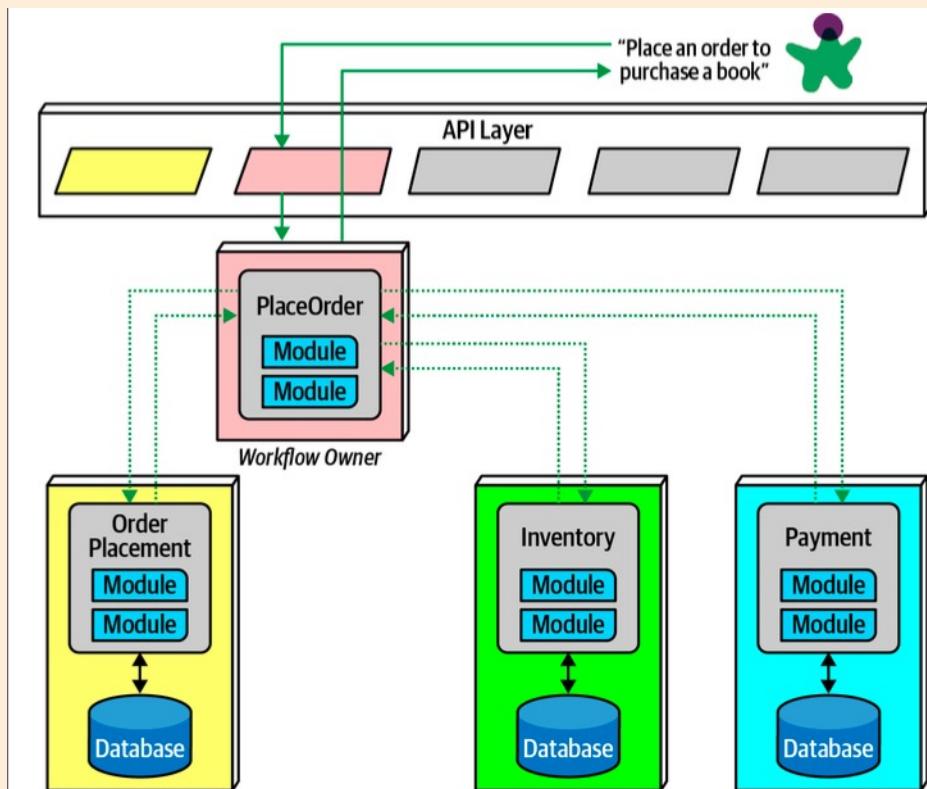
# #Option 1 : Chorégraphie

Chaque service appelle les services nécessaires à la demande. Le principal effet négatif est l'ajout de complexité au niveau du service.



# #Option 2 : Orchestration

Un médiateur local qui va englober la complexité et la coordination. Cela permet à l'architecte de concentrer la coordination sur un seul service, en laissant les autres moins affectés.



# Transaction

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 ?

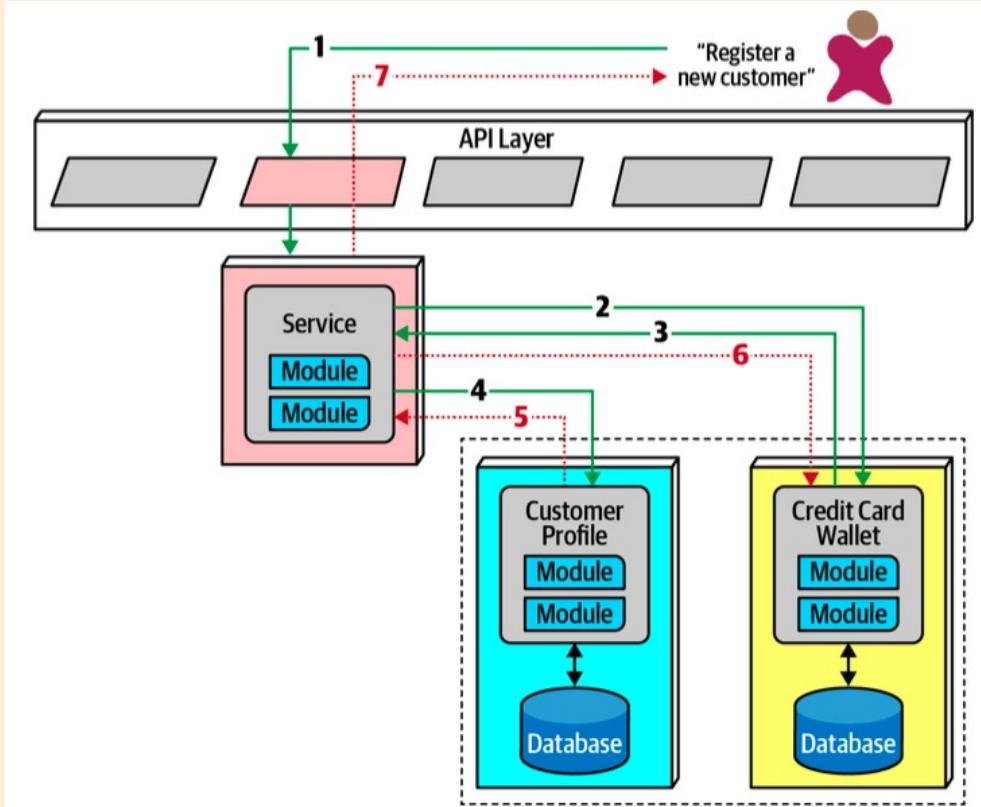
# Transaction : comment faire ?

## 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

# Comparaison d'architectures

# Comparer les architectures

Les connaître c'est bien,

Mais savoir les mettre en opposition, trouver des compromis pour faire le bon choix c'est mieux !!!

La semaine pro ???

# Event-Driven Architecture

(Distribuée)

<https://youtu.be/gOuAqRaDdHA>

# Objectif

Traiter des évènements de manière asynchrone

# Vision traditionnelle



A veut appeler B



A envoie un msg à B



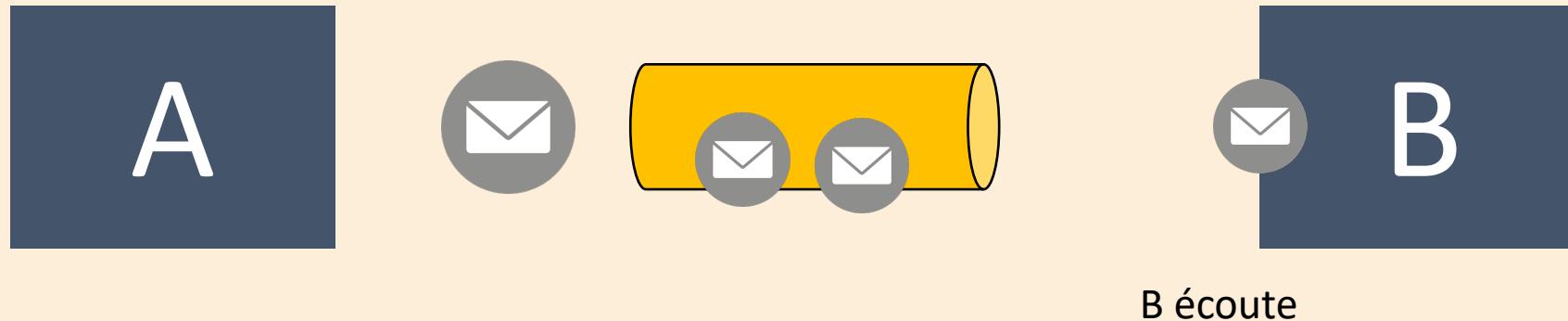
A envoie à la queue

B écoute la queue

**B rend un service à A**

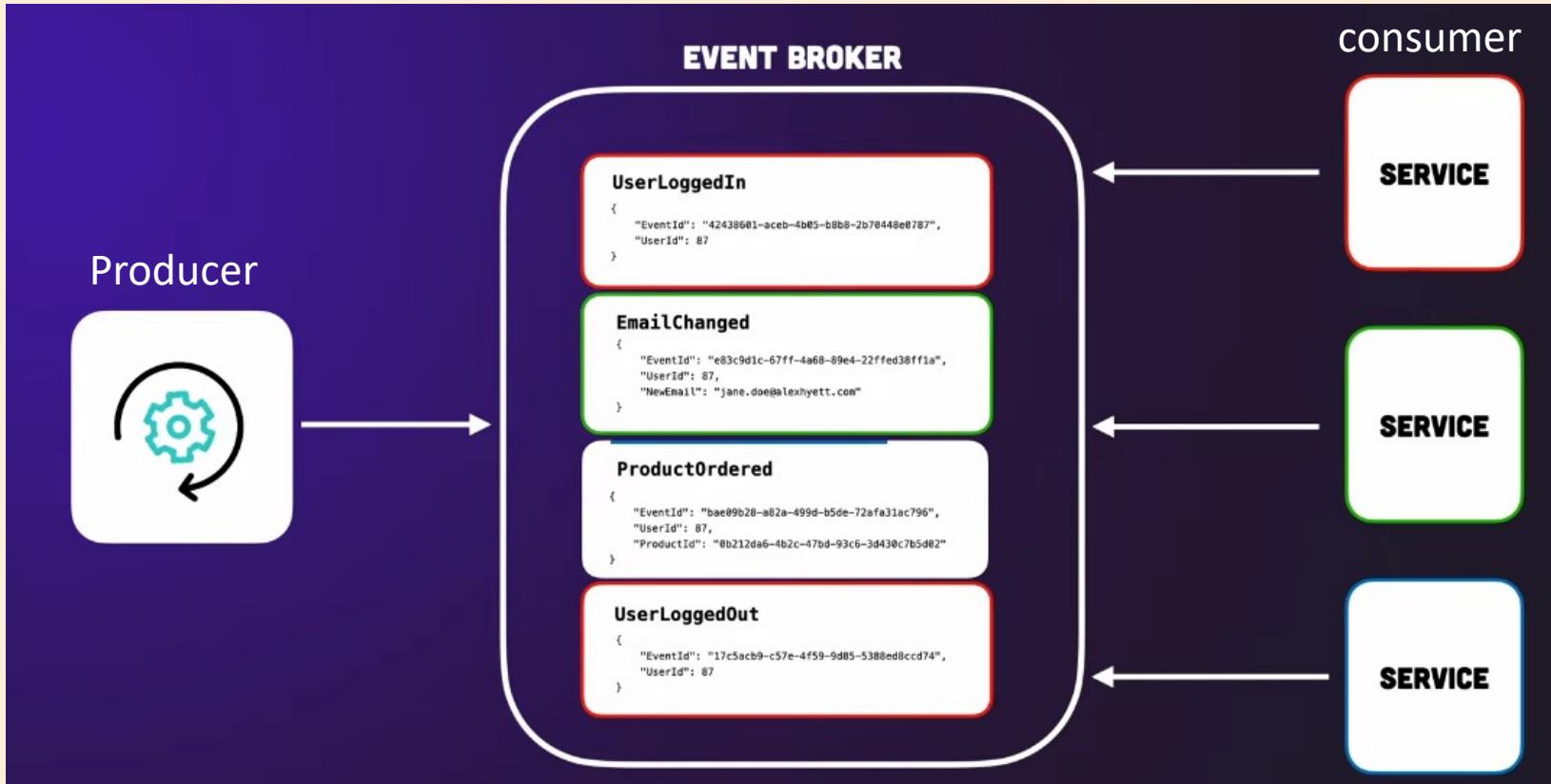
# Vision évènement

B récupère les messages quand il est prêt à les traiter



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

# Proceder/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 leurs

# Quand l'utiliser

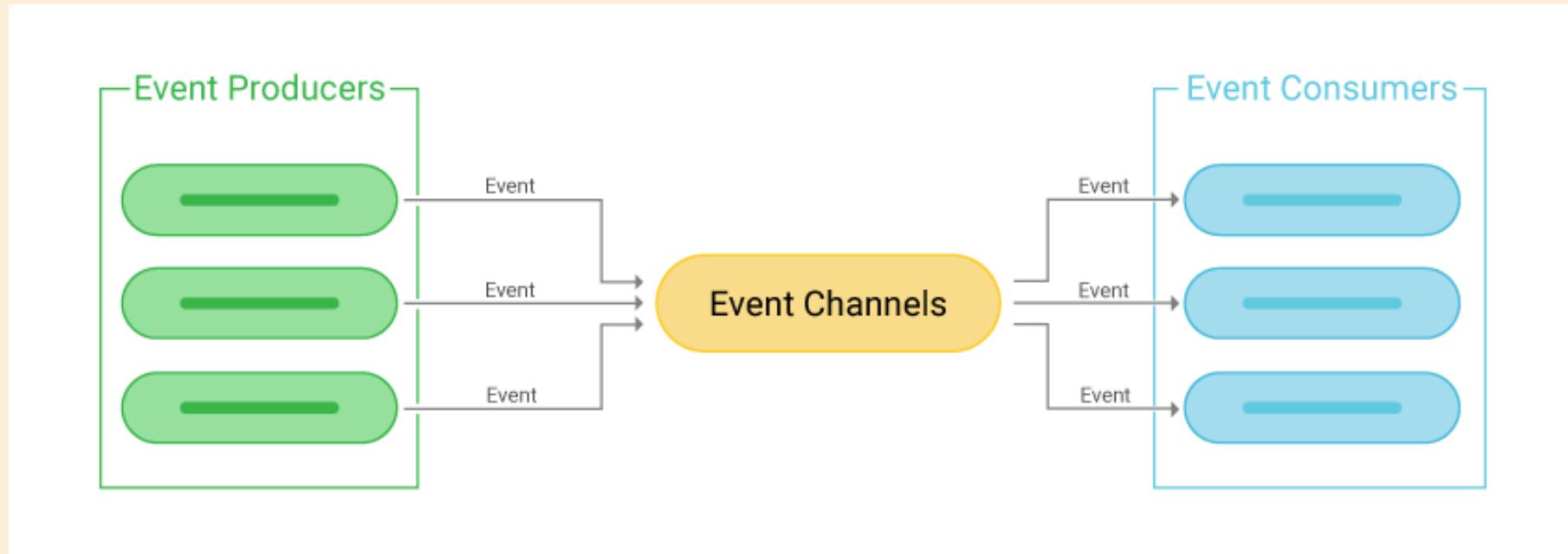
e.g : un système d'email après une commande

- 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.*

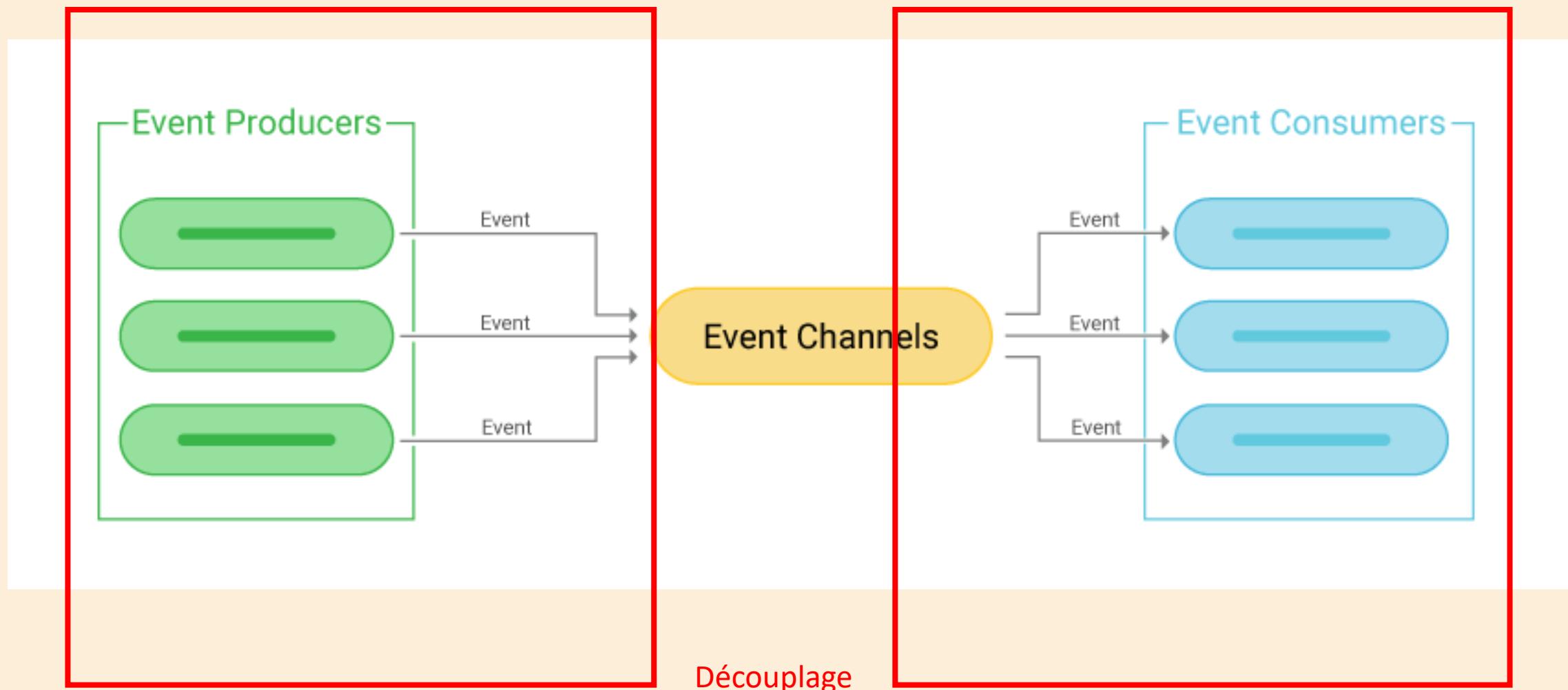
# Avantages



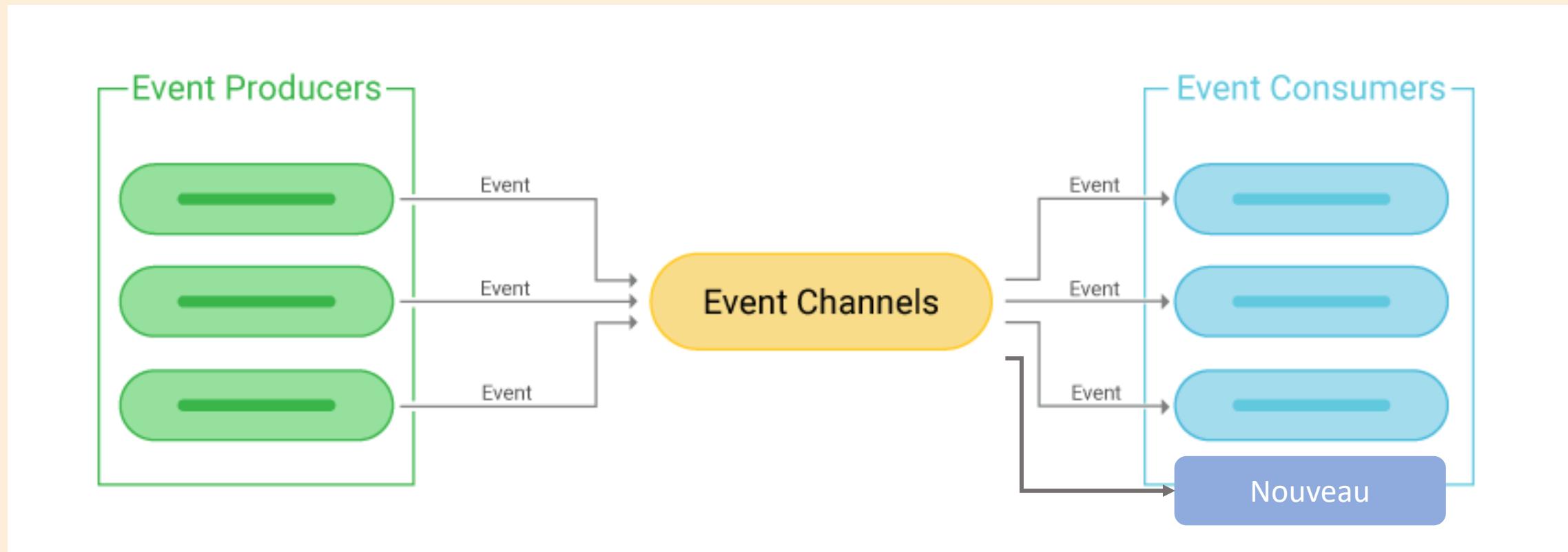
Question :

A votre avis, quels sont les avantages ?

# Avantages

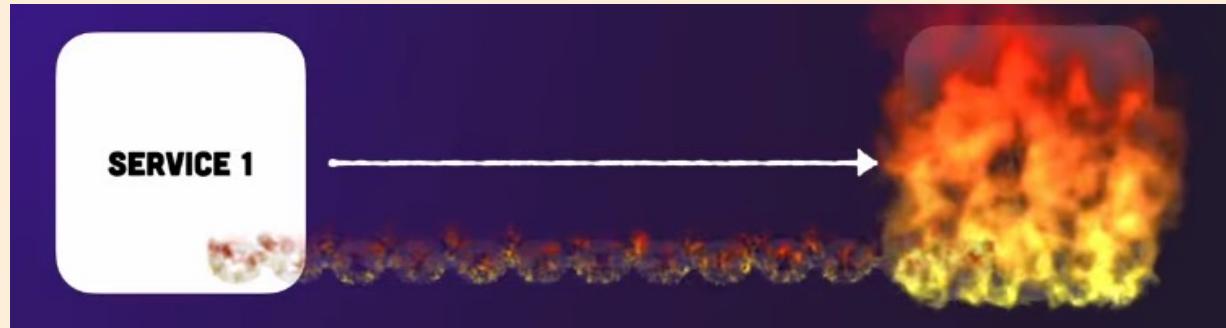


# Avantages

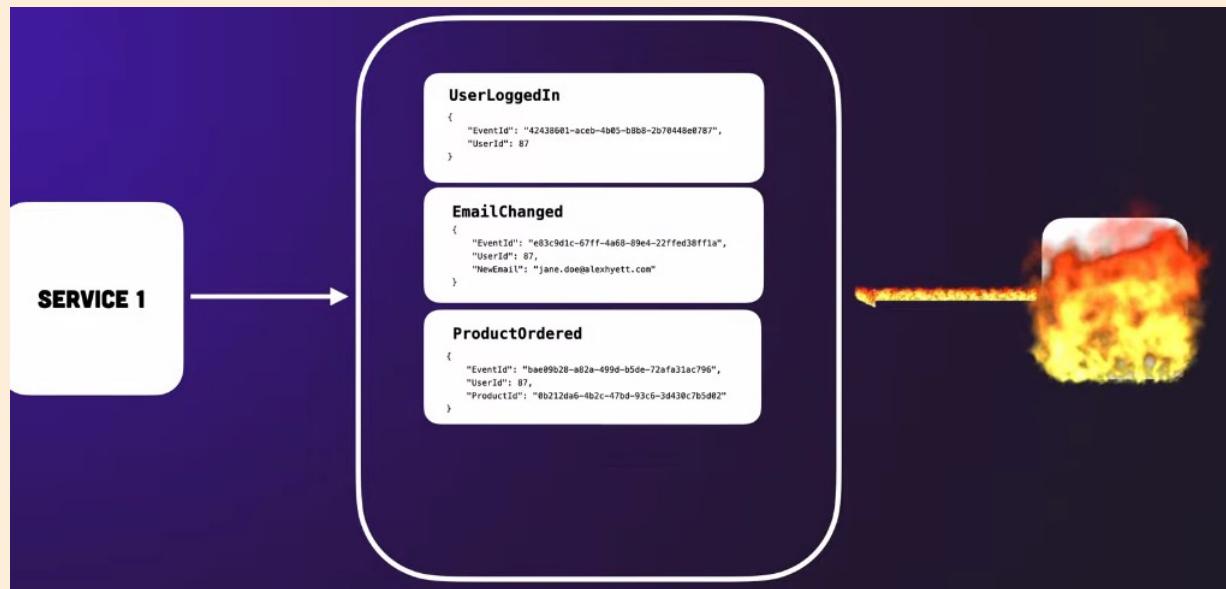


Ajout d'un nouveau consumer facilement

# Avantages



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

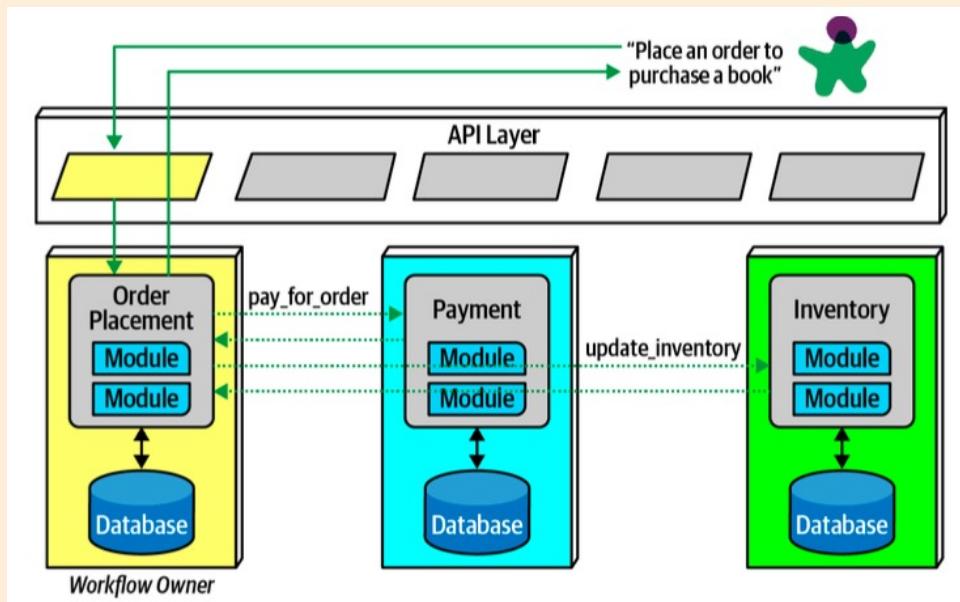


S2 offline, aucun problème  
Le broker stocke les messages

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

# NOTE IMPORTANTE

- Dans les architectures précédentes (microservice, service-based, etc ...)
  - Vous pouvez faire soit du synchrone
  - Soit de l'asynchrone en exploitant des notions de l'event-driven architecture



La communication entre OrderPlacement et Payment peut

- Être synchrone (appel classique de fonction)
- Asynchrone (en utilisant EDA par exemple)

# Comparaison d'architectures

# Comparer les architectures

Les connaître c'est bien,

Mais savoir les mettre en opposition, trouver des compromis pour faire le bon choix c'est mieux !!!

La semaine pro ???

# C'est la fin

De la première section ...

# Caractéristiques architecturales

4

# Objectif de la section

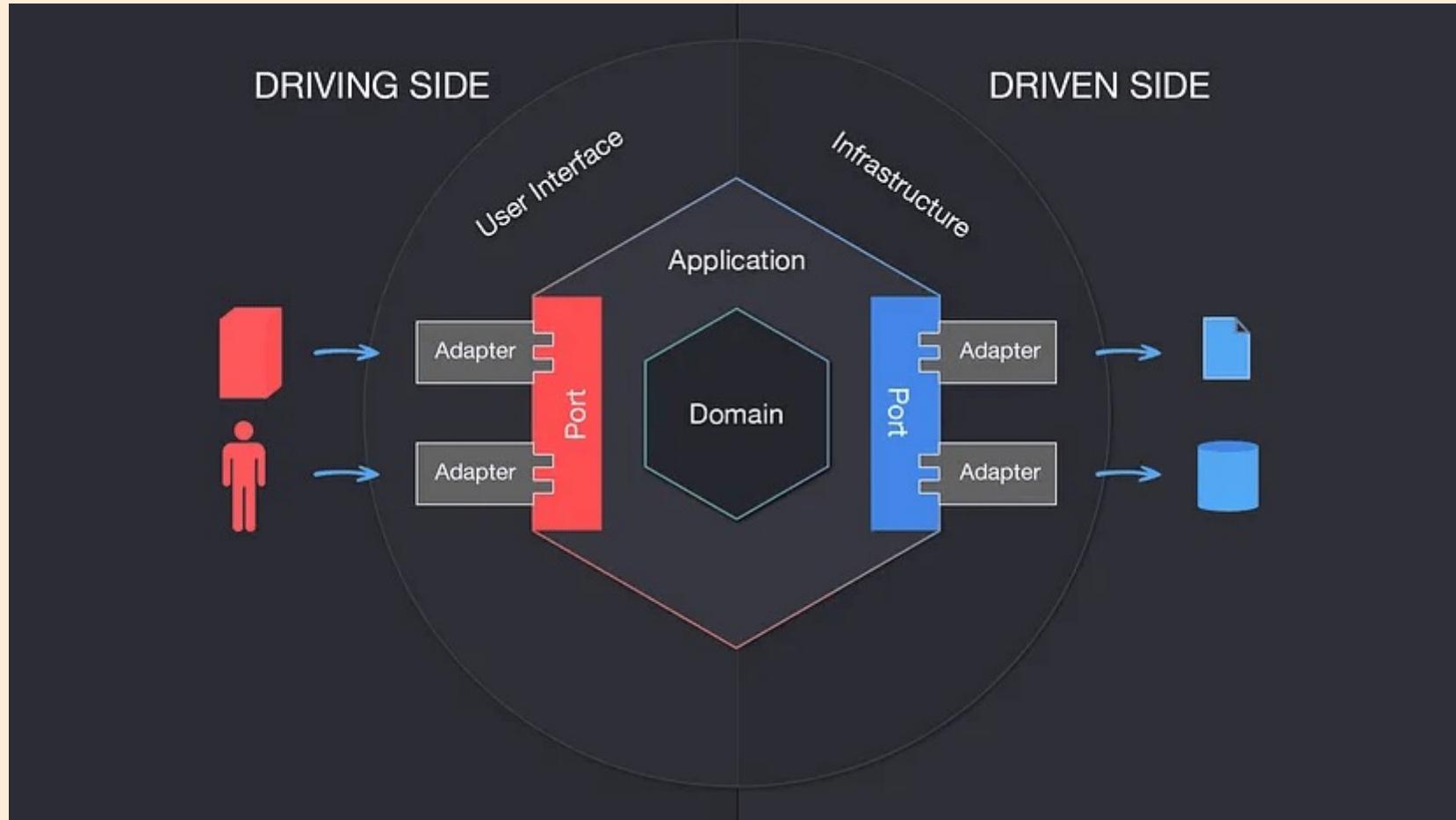
- Nous avons vu les styles architecturaux (monolithique et distribué)
- Dans cette section, nous allons étudier quelques patrons architecturaux fort utile
  - L'objectif est d'être un peu plus concret avec des diagrammes de classe (mais sans code; **si vous voulez le code on le fera**)

# Hexagonal Architecture

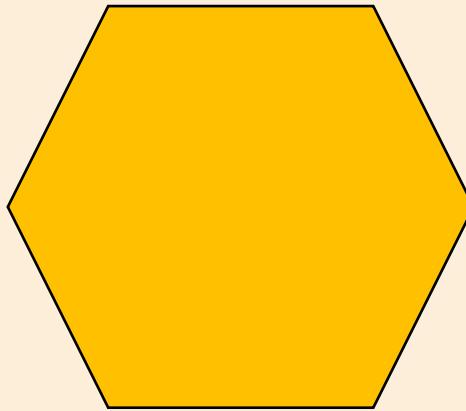
<https://youtu.be/bDWApqAUjEI>

## Question :

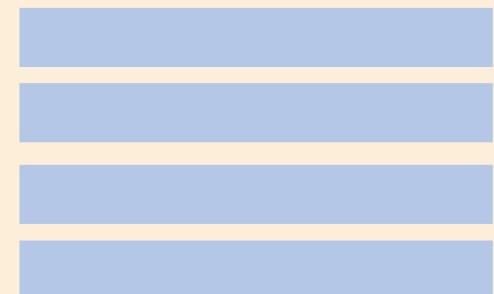
Que pouvez vous me dire ?



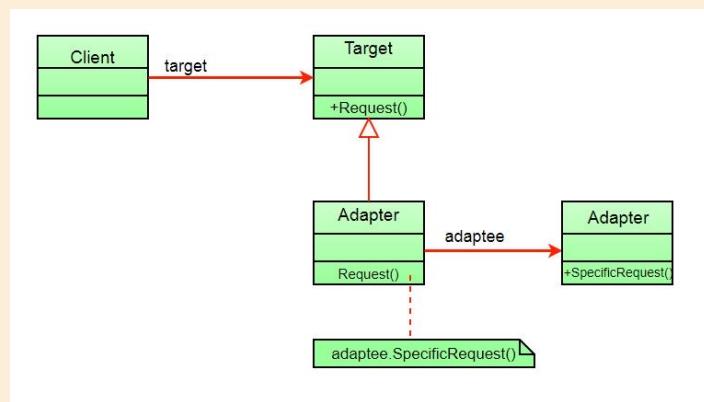
Hexagonale architecture



Layered / n-tiers architecture

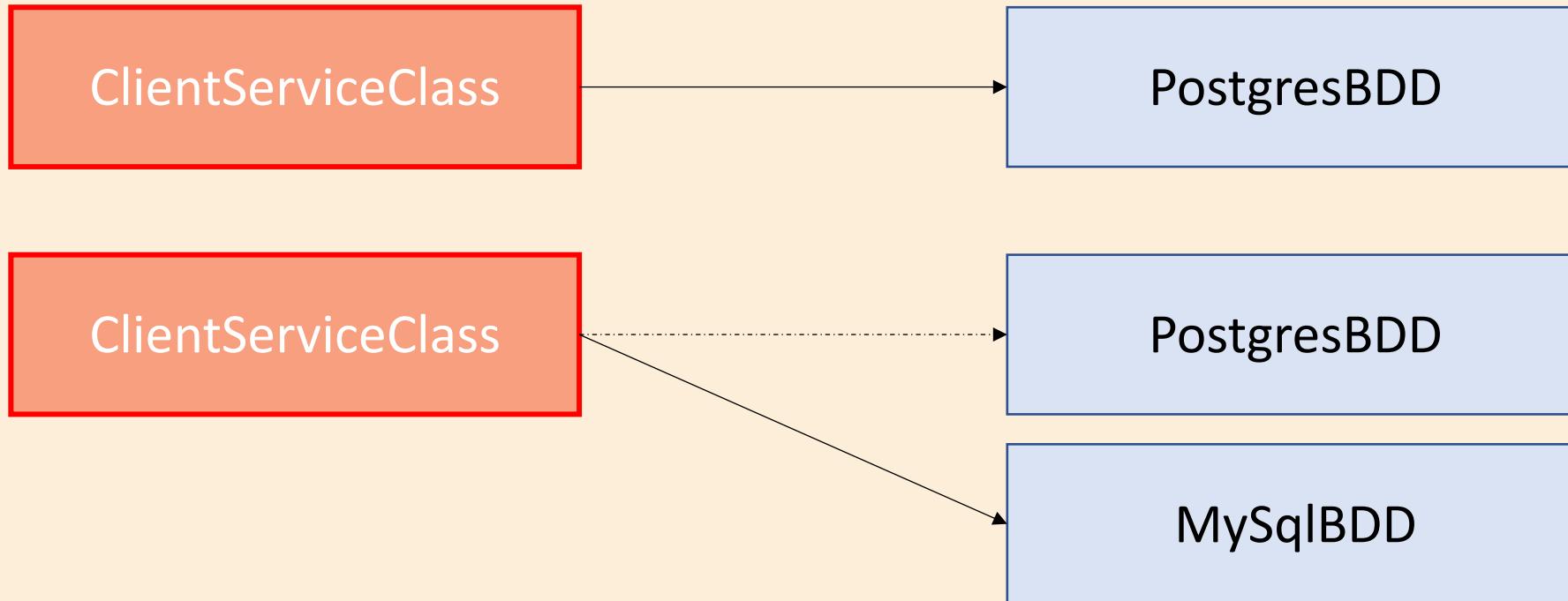


Communication entre couches  
Via DP Adaptateur



**Vers** une source externe  
(output)

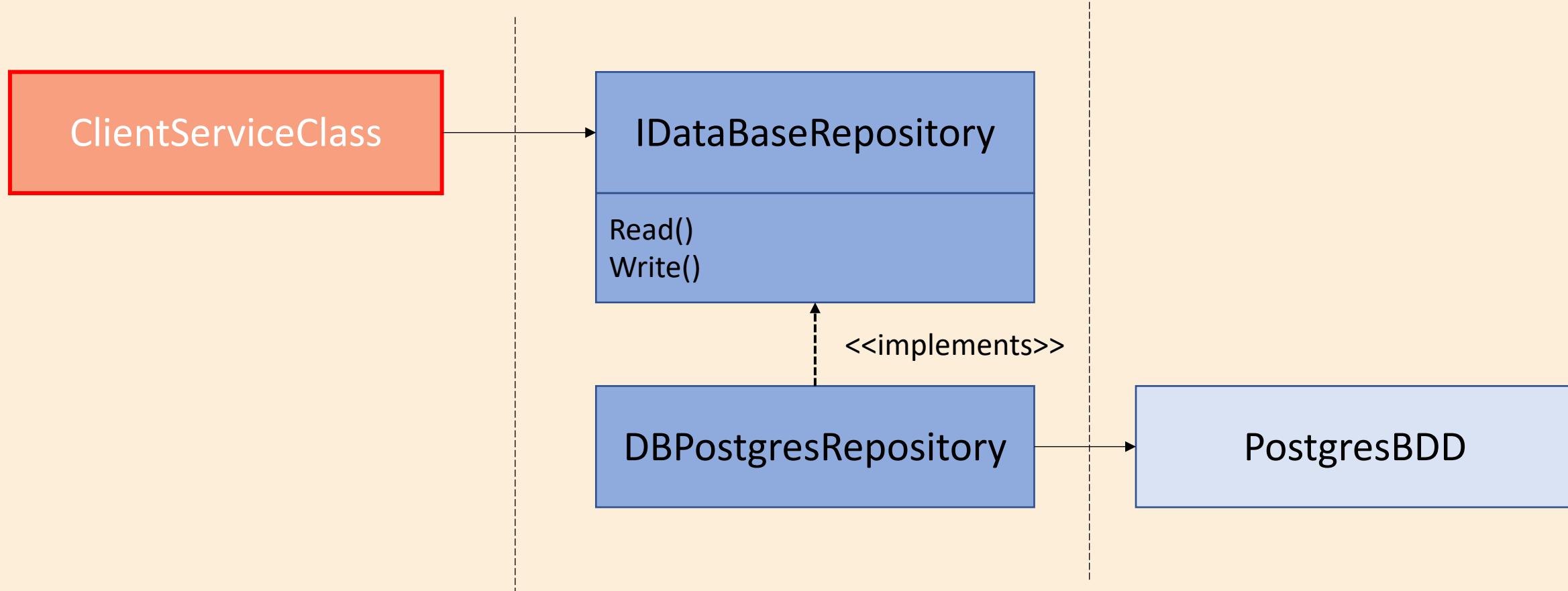
# Le problème



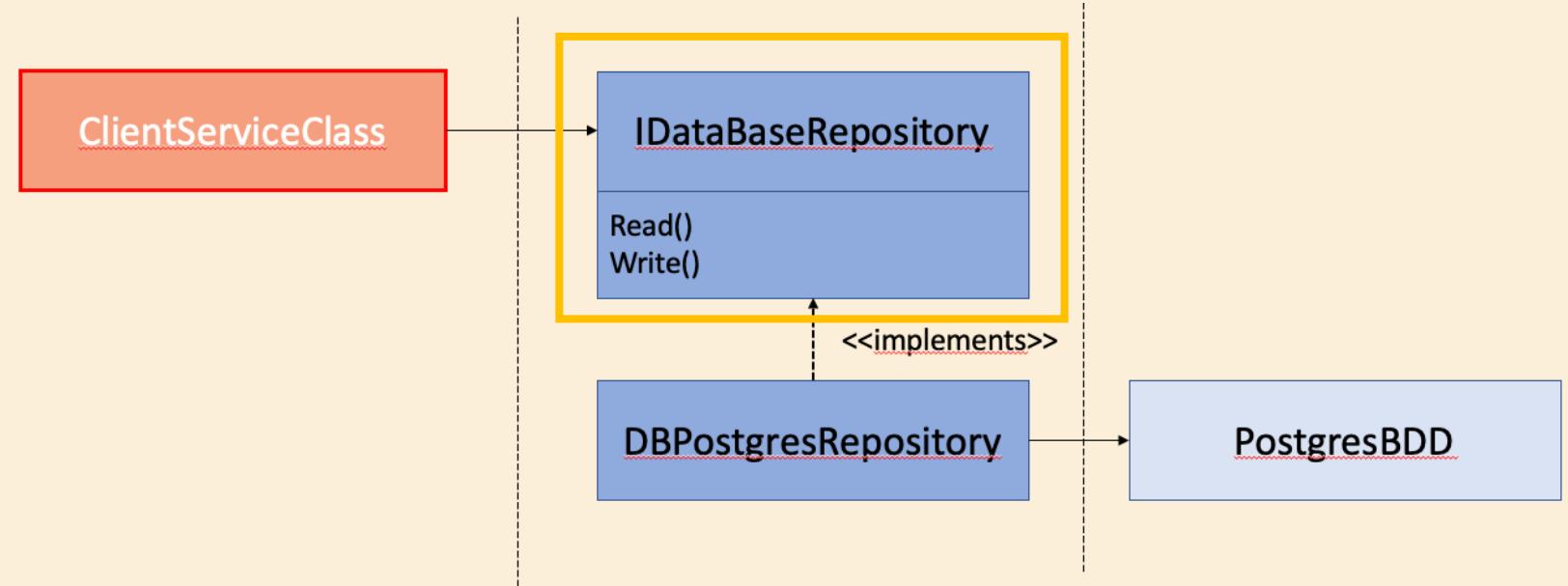
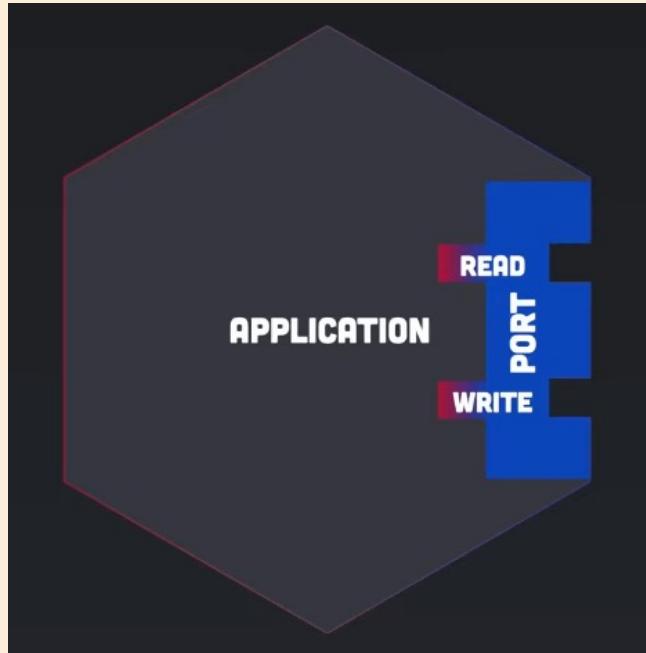
Je veux maintenant utiliser une base de donnée MySQL, je vais devoir recoder ClientService

# La solution

Au lieu d'appeler directement la BDD, on va passer par un élément intermédiaire

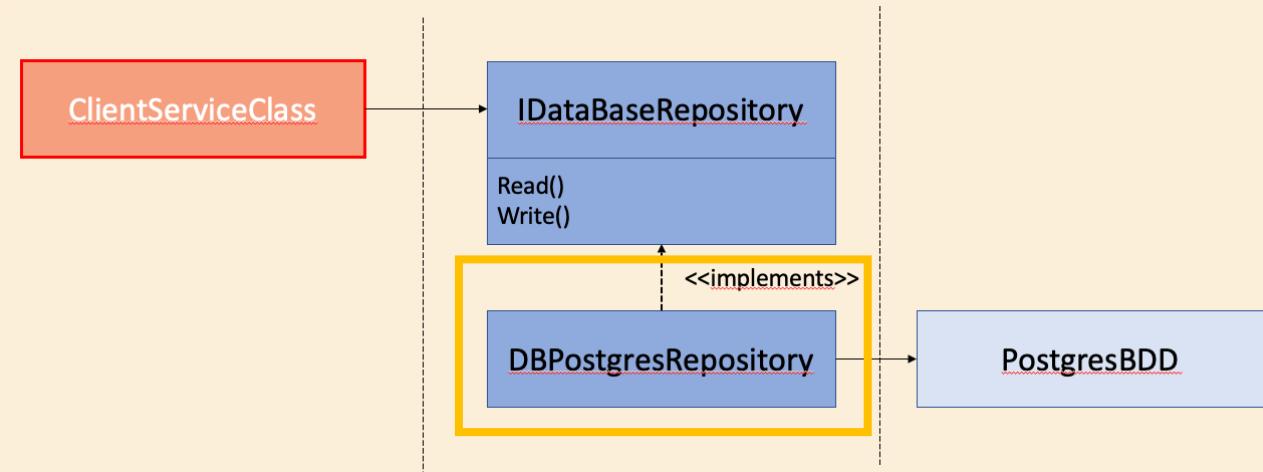
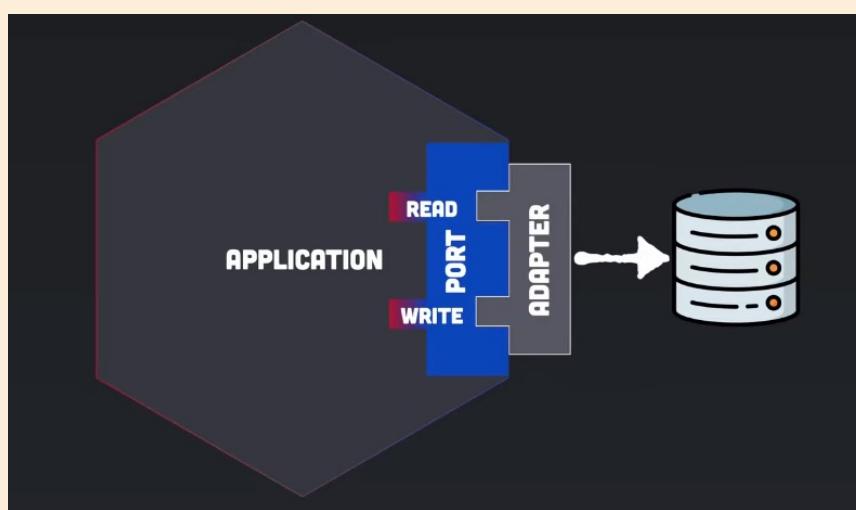


# La solution : le port

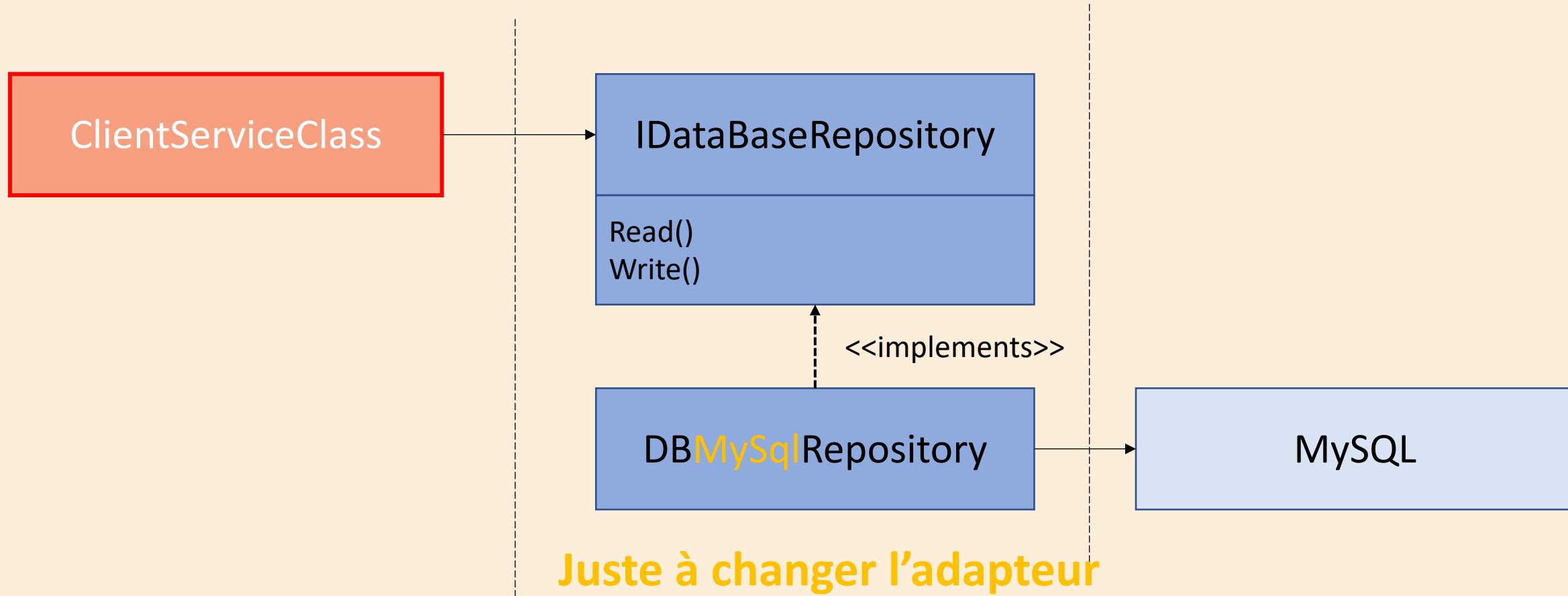


C'est une interface définie par nous dans notre application

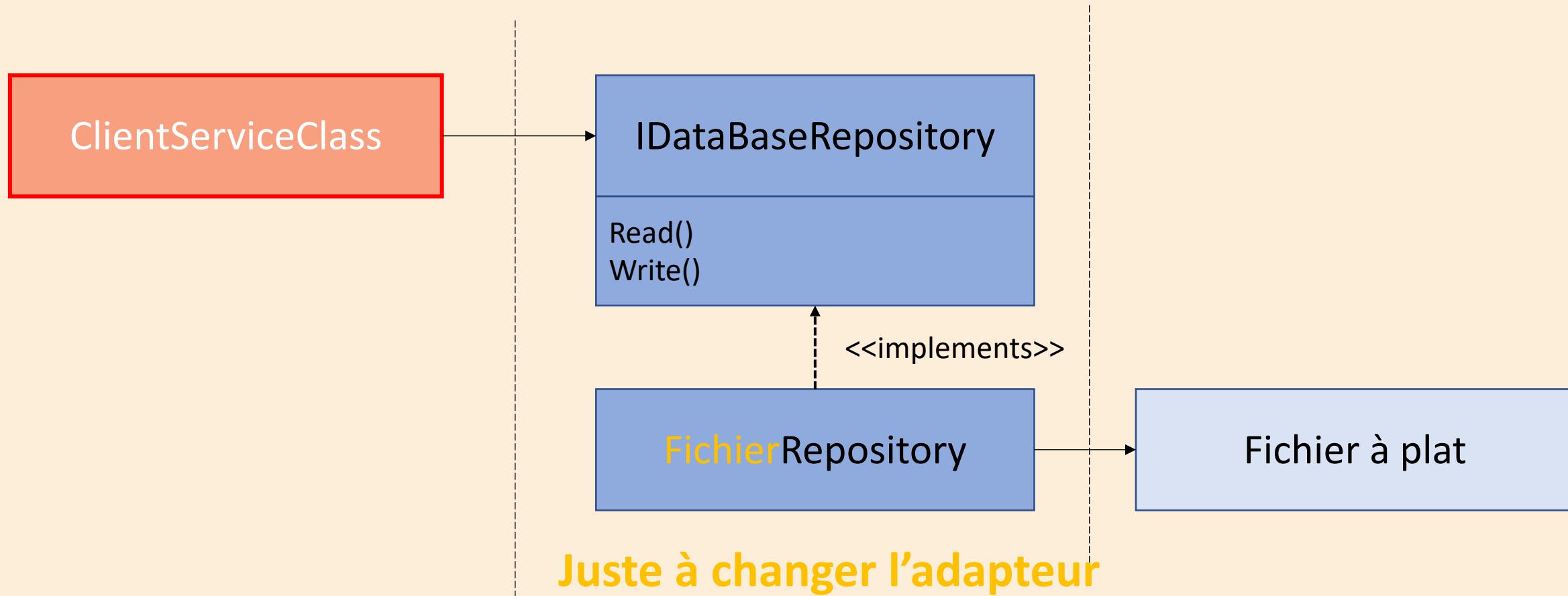
# La solution : adapteur



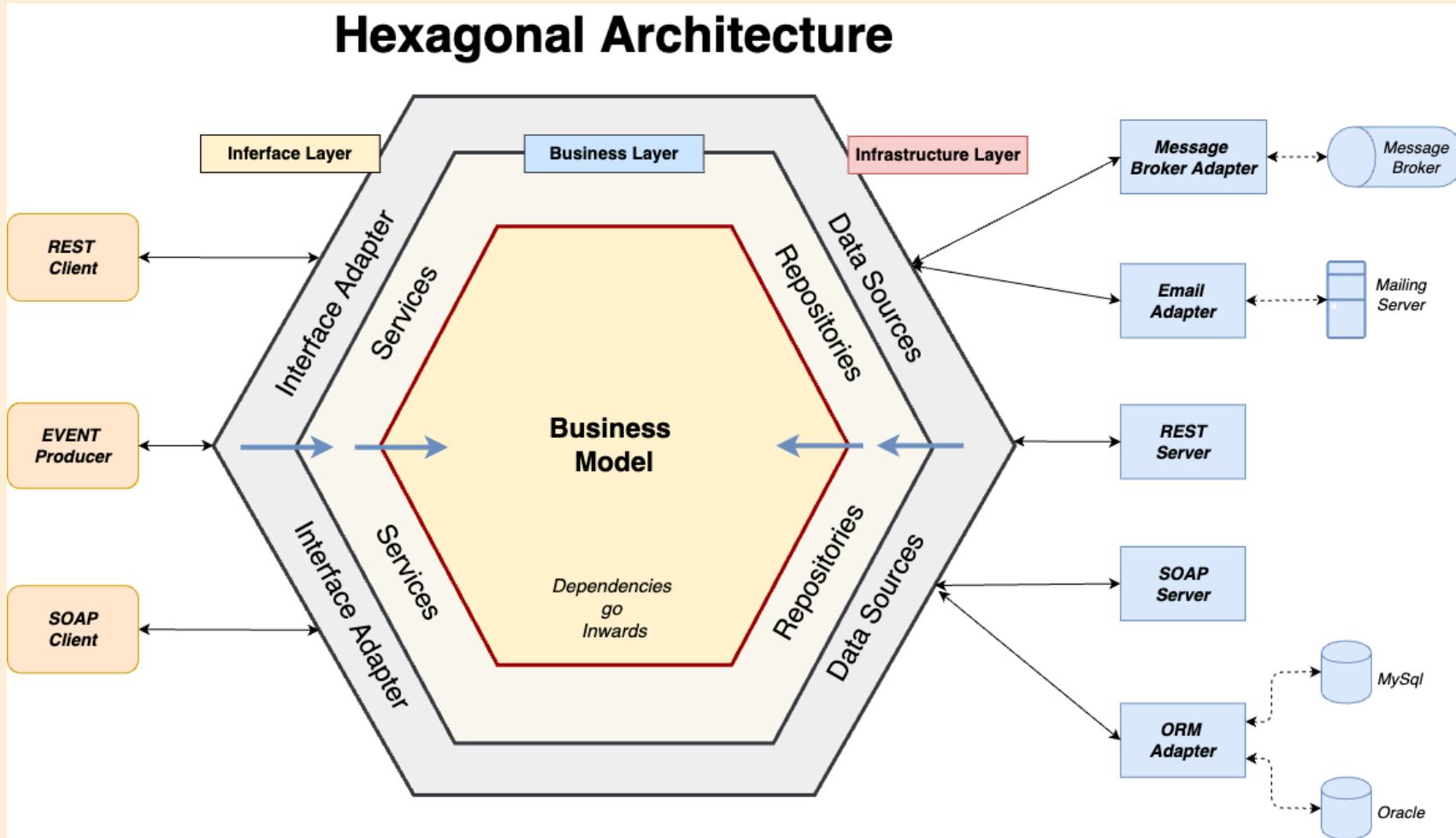
# La solution : vers MySQL



# La solution : vers Fichiers

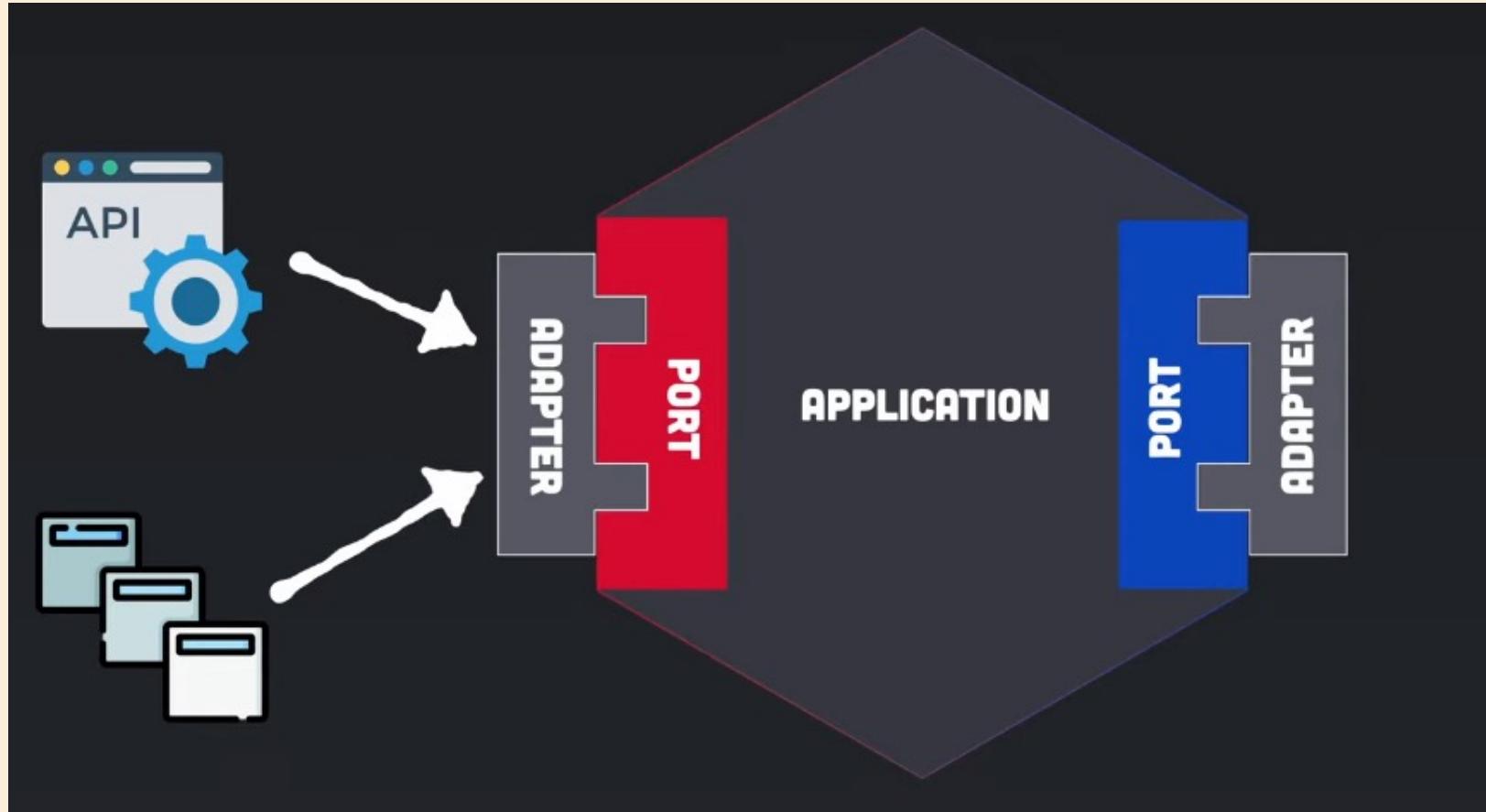


# Résumé

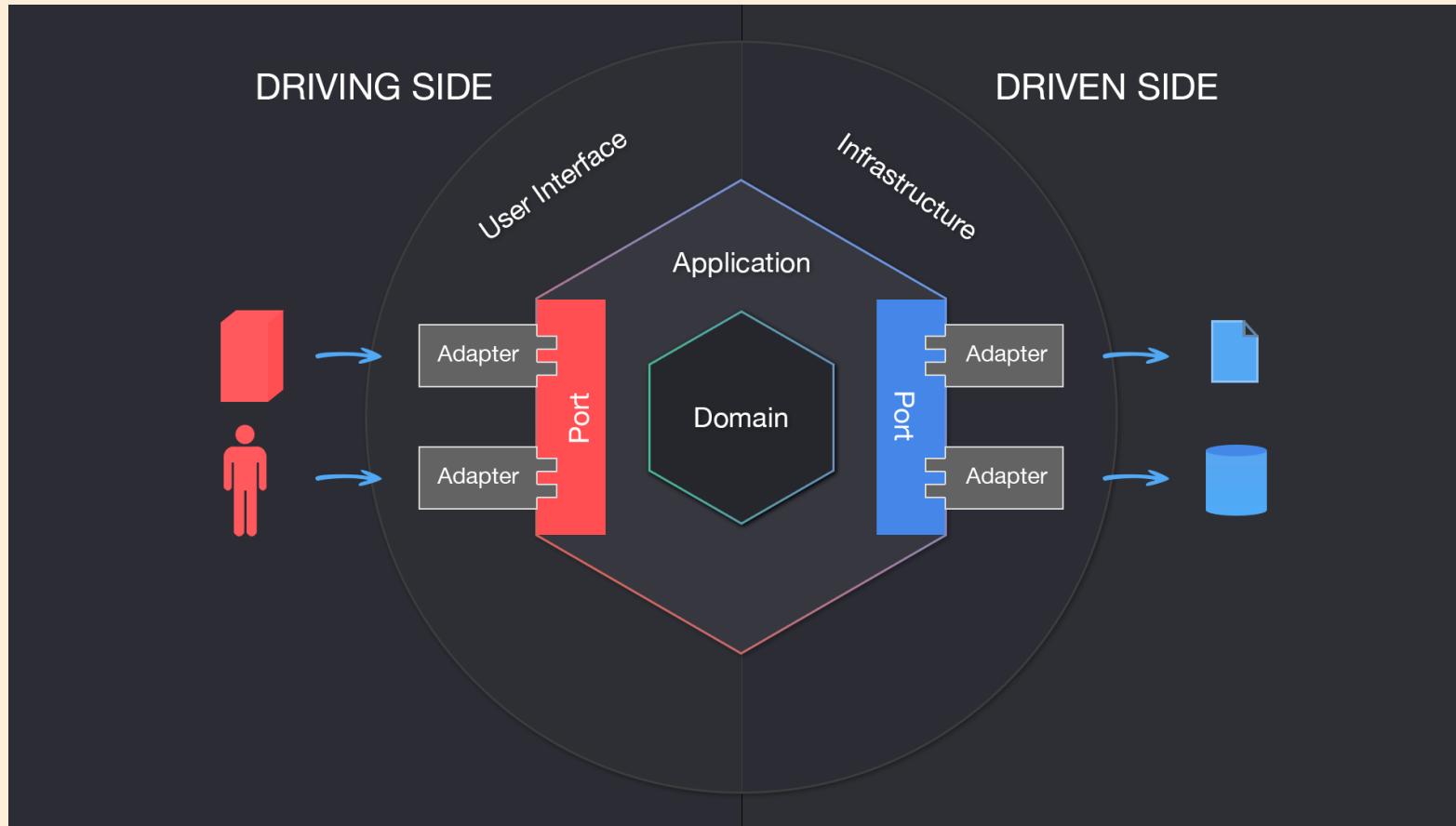


Depuis une source externe  
(input)

# Le même principe



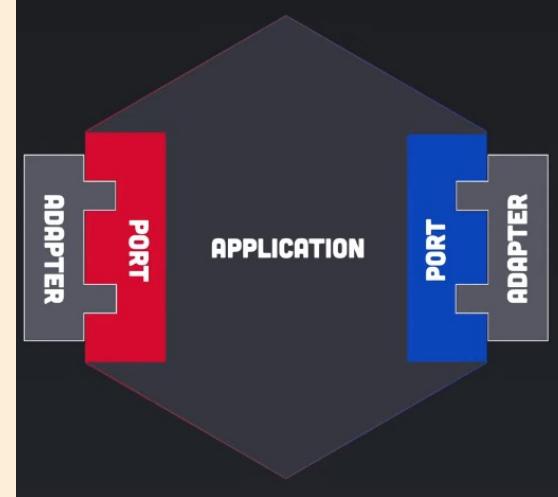
# Vue complète (driving et driven side)



# Les avantages

# Testable

- Le domaine est totalement indépendant
- Il fournit des prises (port) et les outils externes s'adaptent  
=> Les ports font abstraction

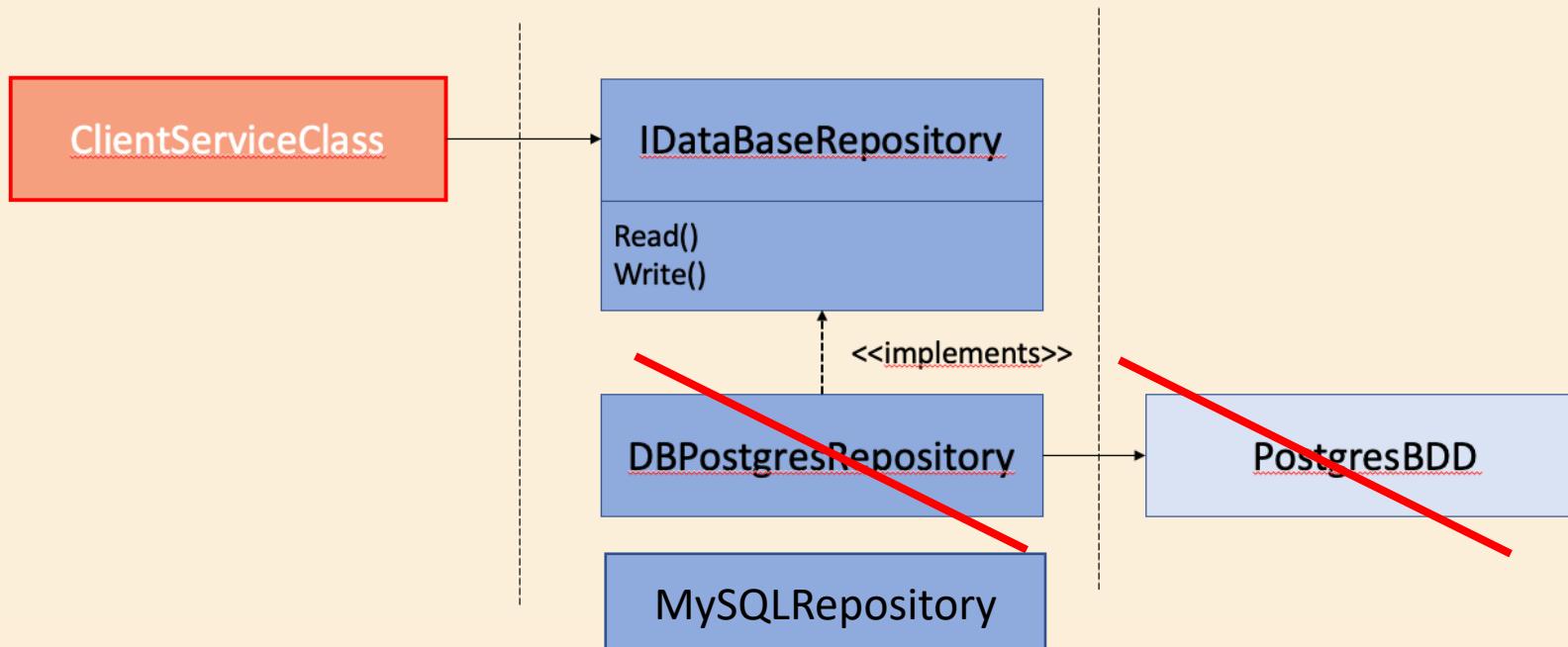


Note :

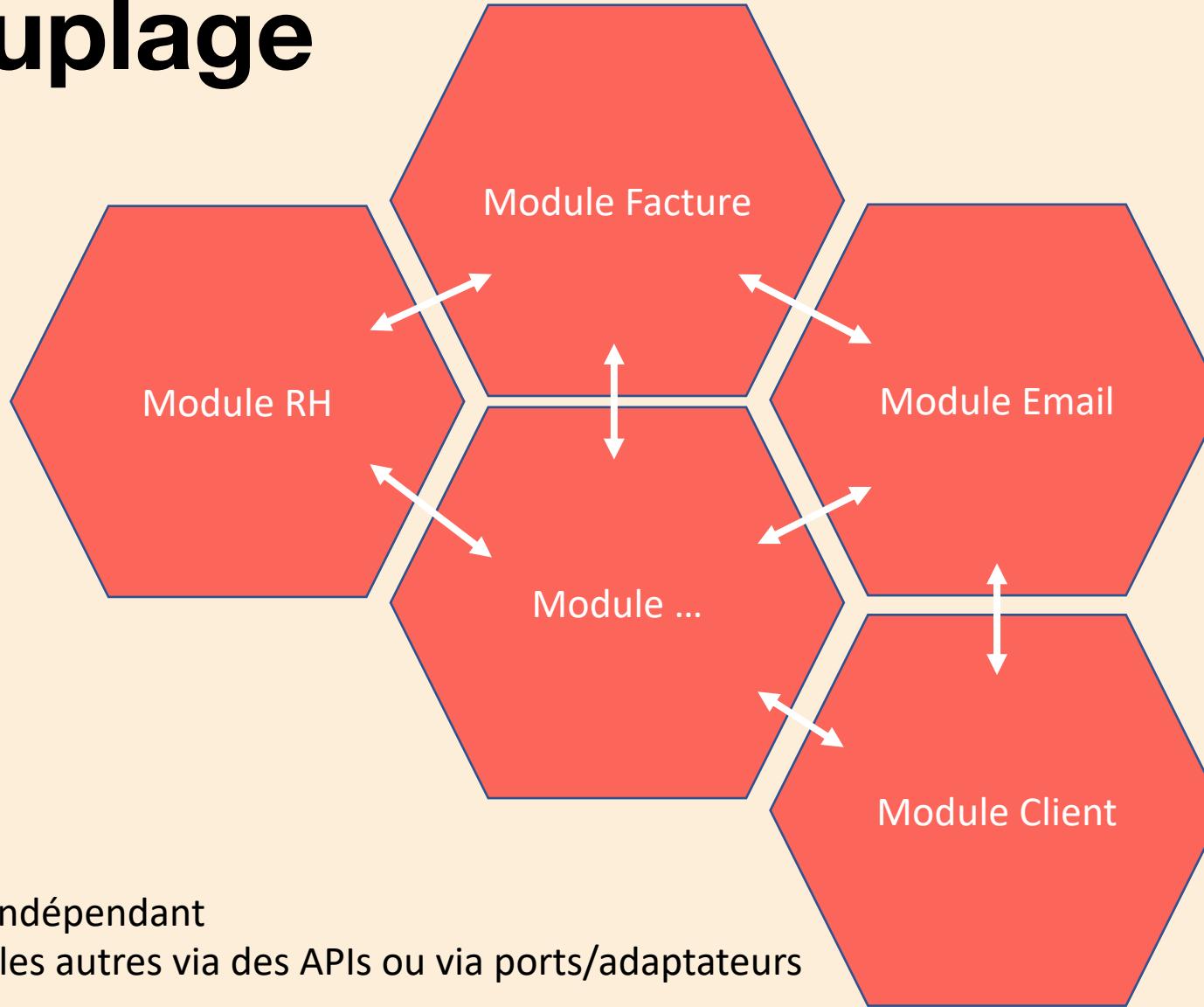
- Il faut coder les ports (les interfaces)
- Et des fois les adapteurs (qui implémentent les ports)

# Mainenable / Flexible

- Si PostgreSQL devient obsolète je pourrai facilement le remplacer, il faudra juste recoder l'adaptateur ... (et pas tout le domaine)

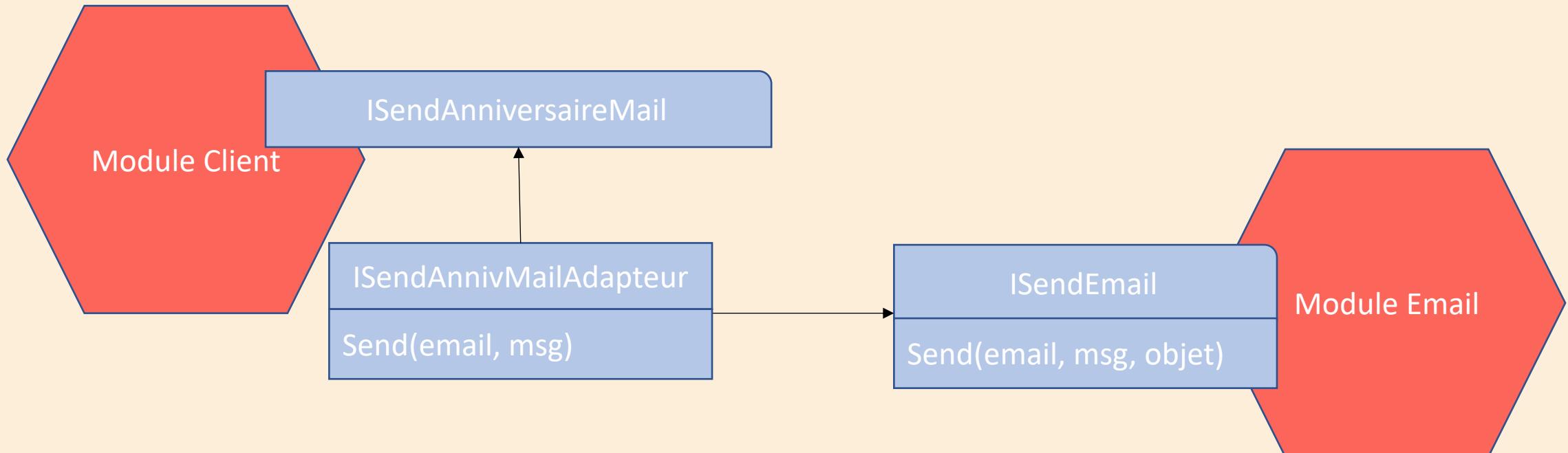


# Découplage



Chaque module est indépendant  
Il communique avec les autres via des APIs ou via ports/adaptateurs

# Découplage



```
● ● ●  
public void send(String email, String msg) {  
    iSendEmail.send(email, msg, "bon anniversaire");  
}
```

# Enterprise Service Bus (ESB)

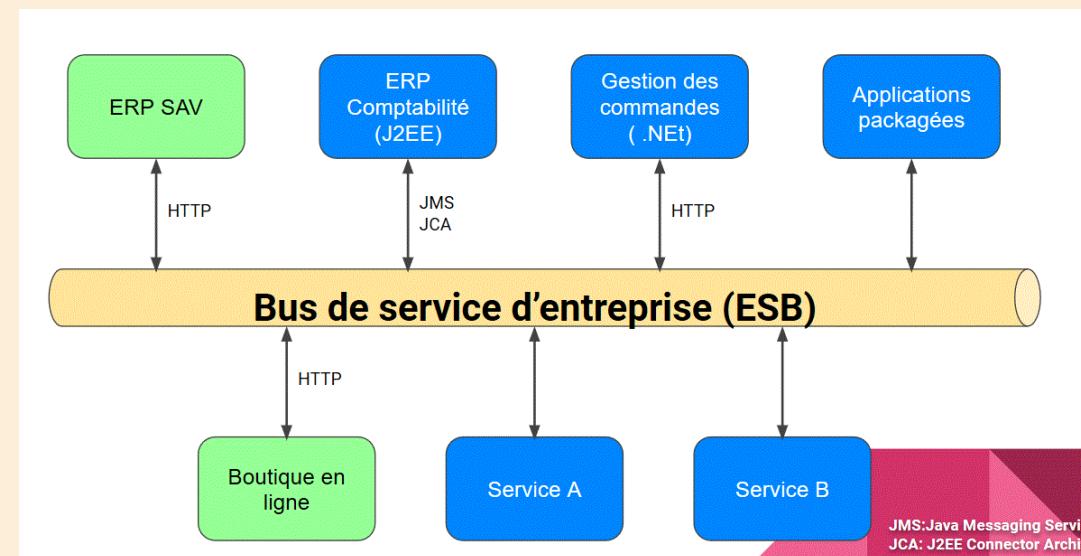
Déjà vu dans SOA

# Lien avec

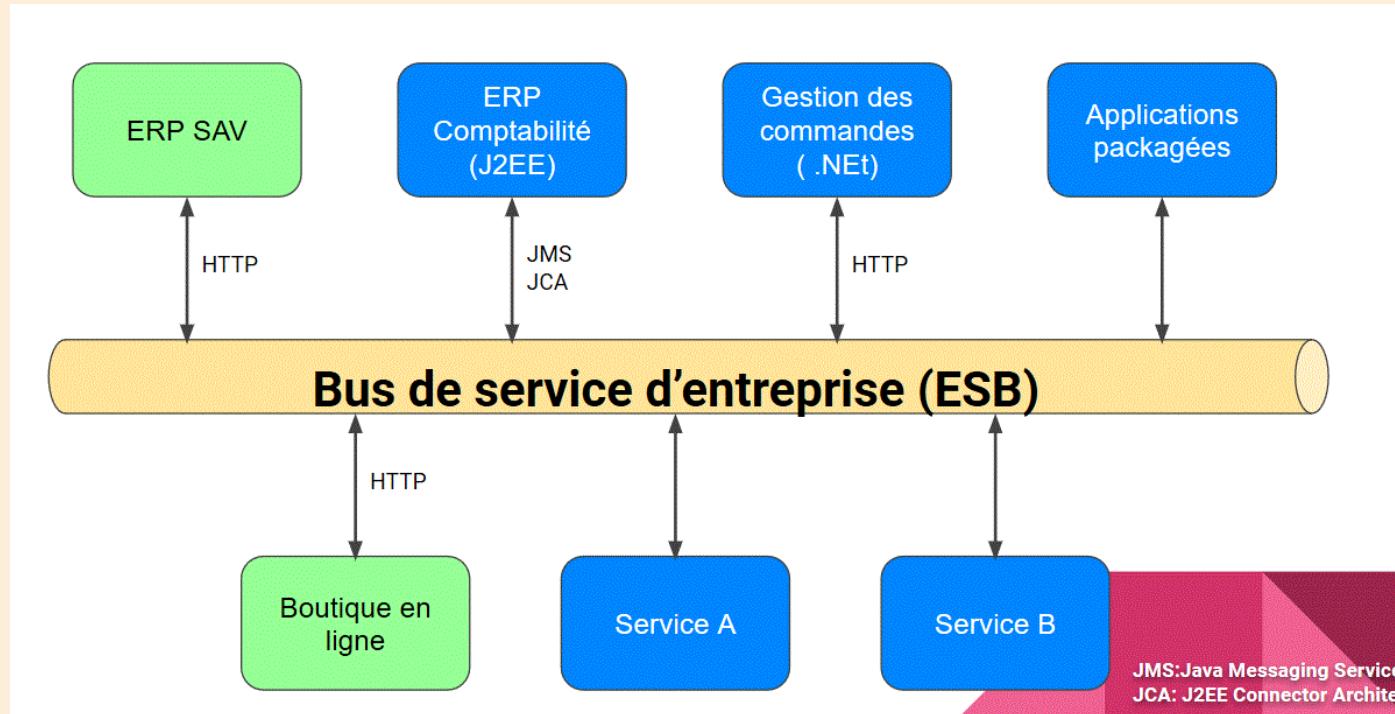
- L'architecture SOA
- L'architecture Microservices

# ESB : Objectif

- Permettre la communication des applications qui n'ont pas été conçues pour fonctionner ensemble
  - E.g : deux progiciels provenant d'éditeurs différents



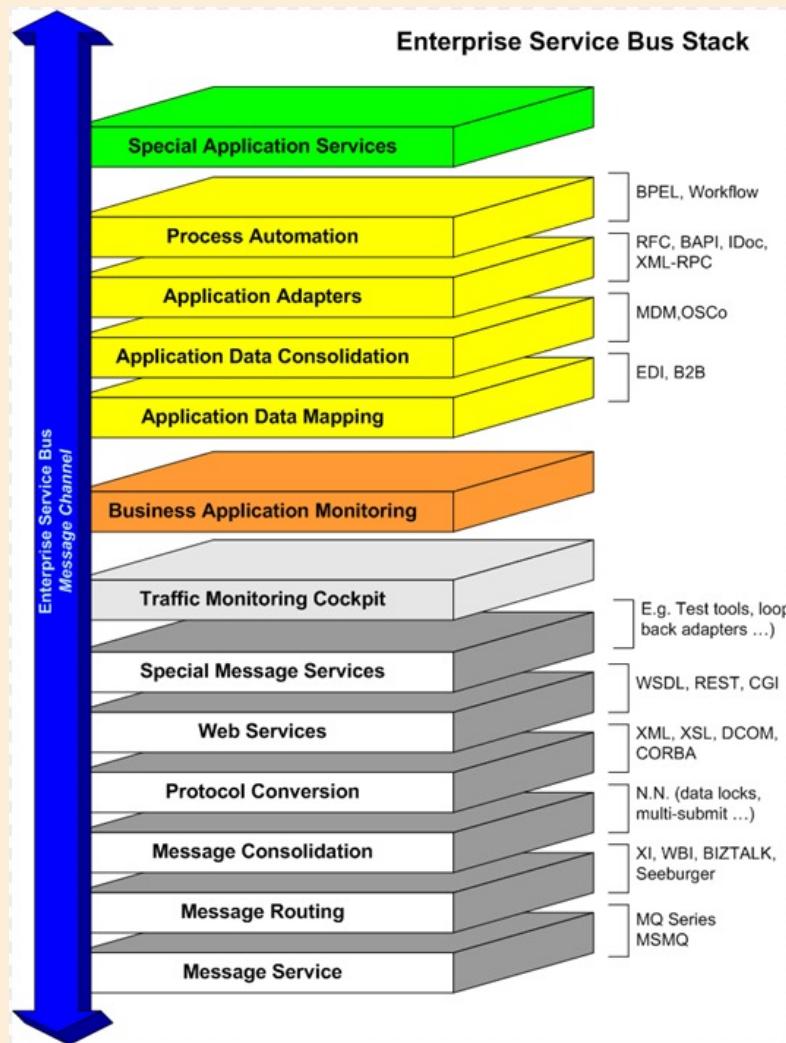
# ESB : architecture



## L'ESB

- Centralise les informations et répartit le travail entre les applications qui sont reliées à lui
- Route les messages reçus. A l'envoi, l'ESB le route vers le destinataire
- Transforme le message pour qu'il puisse être lu par le destinataire

# ESB : composition



# Extract, Transform, Load (ETL)

[https://youtu.be/\\_QP-Kblt61U](https://youtu.be/_QP-Kblt61U)

# Data-pipeline

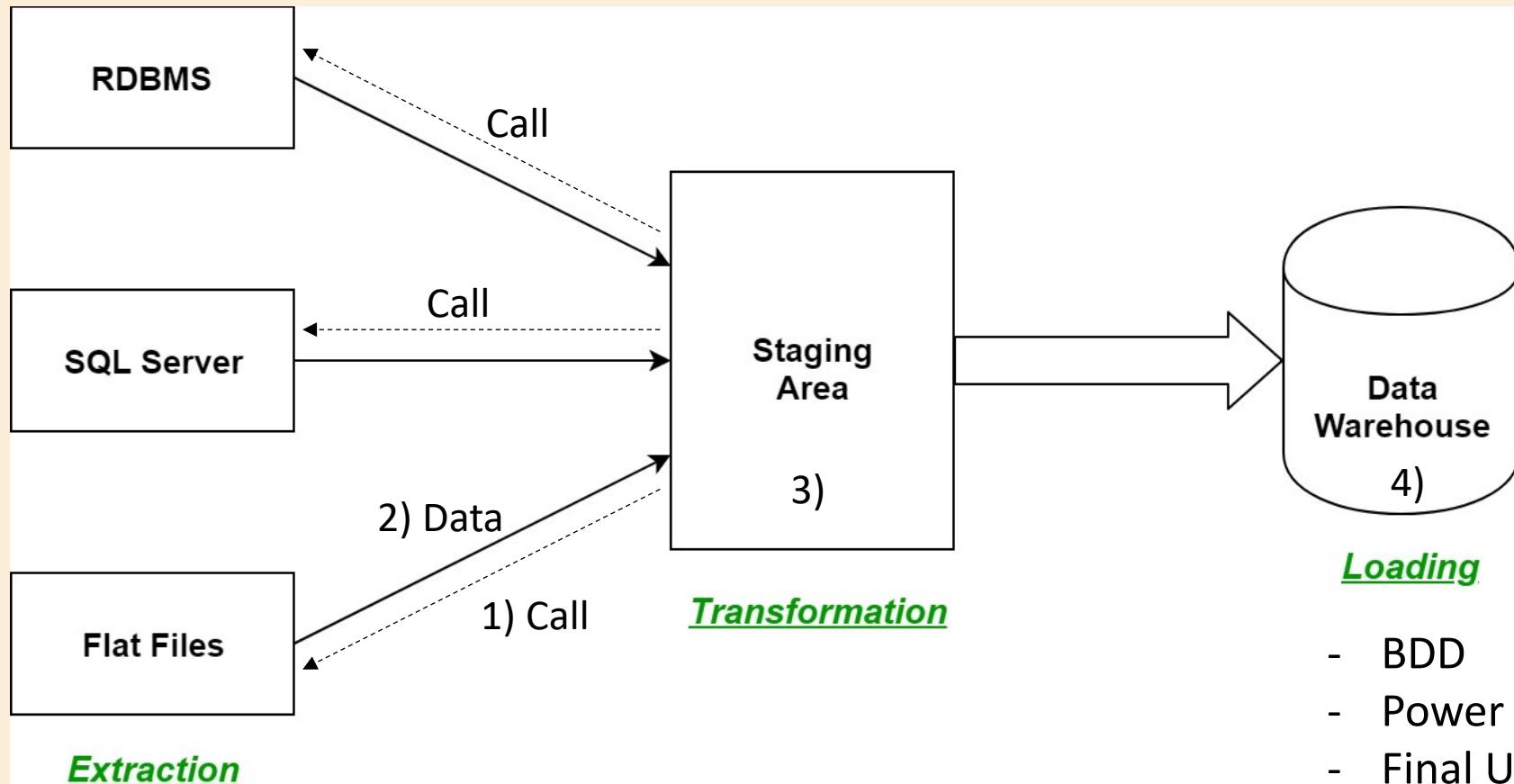
- Les entreprises gèrent des données
- Une data-pipeline est une série de processus qui déplacent des données d'un système à un autre.

# Data-pipeline : cas basique



- Chaque étape génère une sortie qui sert d'entrée à l'étape suivante
- E.g :
  - Data est extraite depuis une source externe (API)
  - Elle est transformée
  - Pour être stockée en BDD (destination) pour pouvoir être analysées

# Extract Transform Load



- BDD
- Power Bi (pour analyse)
- Final User

# # Extract

- Extraire les données depuis plusieurs sources
  - Fichier
  - BDD
  - API
  - Object connecté

# # Transform

1. Nettoyer les données
2. Standardiser les données
  - Définir un format et un mode de stockage commun
3. Supprimer les doublons et anomalie



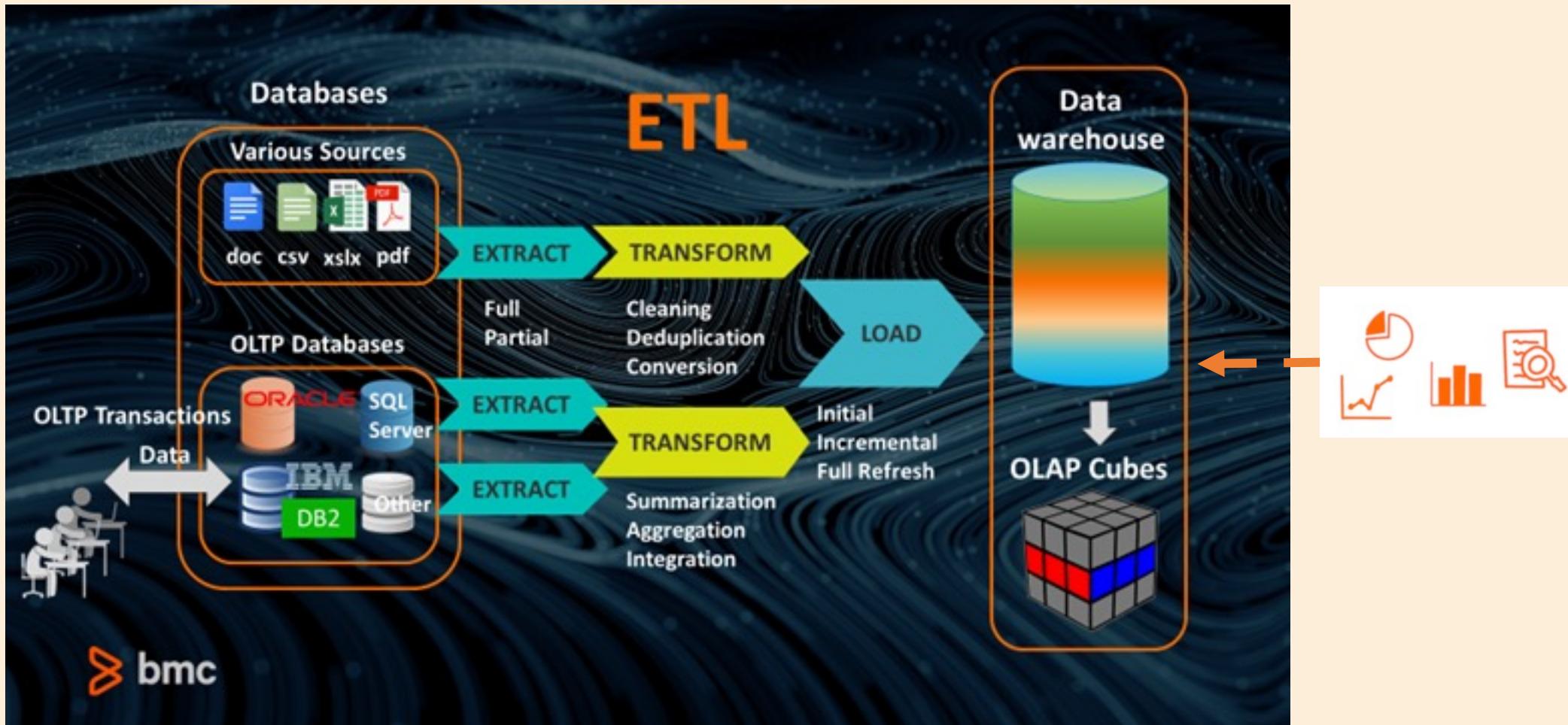
**Les données sont maintenant dans un format exploitable**

En Machine Learning on effectue ces étapes avant de construire notre modèle  
« Data Preparation »

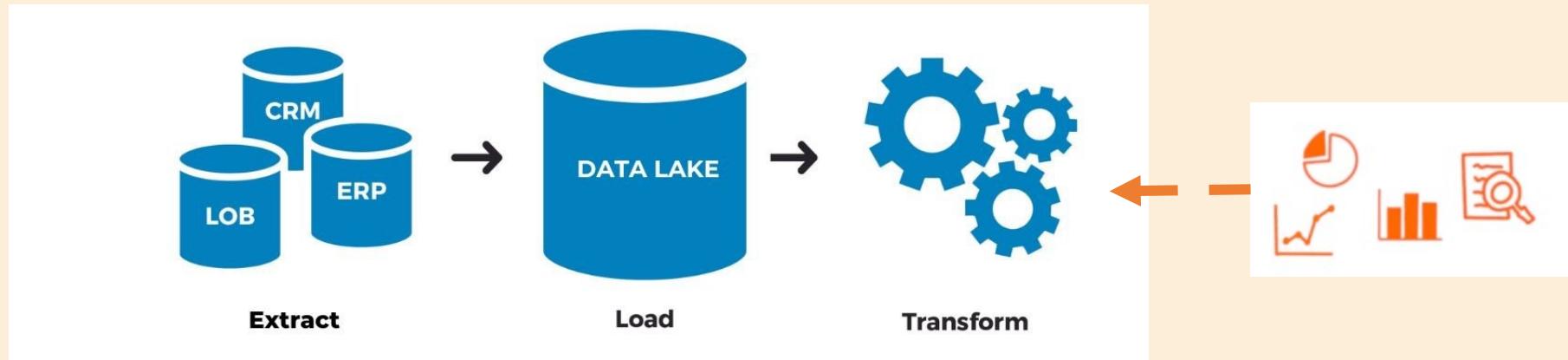
# # Load

- Charger des données vers un stockage cible
  - Souvent un datawarehouse
- **Les données (formalisées) sont donc centralisées à un même endroit**
- Les données sont maintenant accessible (appel d'une API) par des analystes pour faire des rapports (Business Intelligence)

# ETS résumé



# Extract Load Transform



1. Vous extrayez des données brutes de différentes sources
2. Vous les chargez dans leur état naturel dans un entrepôt de données ou un lac de données
3. Vous les transformez selon vos besoins dans le système cible

# ETL VS ELT

- Avec un ETL vous n'avez pas accès aux données brutes (elles sont perdues) durant la transformation
- Avec un ELT vous avez accès aux données brutes et pouvez travailler dessus (nettoyer, formaliser) autant de fois que vous voulez
- La mise à disposition des données (cas Big Data) est plus rapide avec un ELT qu'un ETL (pas de transformation). La transformation est faite à la demande