

# Architectures logicielles

Nicolas Ferry

*Université Côte d'Azur - IUT Informatique - LP IOTIA*

**2022 - 2023**

# Objectif du cours

---

- Qu'est ce qu'une **architecture logicielle**.
- Vous introduire les **grands types** d'architectures logicielles :
  - Les **concepts** de ces architectures.
  - Comprendre les **atouts et inconvénients** de ces architectures.

# Contenu

---

- 1. Introduction aux architecture logicielles**
- 2. Architecture N-Tiers**
- 3. Event Driven Architecture**
- 4. Architecture orienté services**
- 5. (Vers les micro-services)**

# Evaluation

---

- **TPs notés :**
  - Sélectionnés aléatoirement.
- **Un contrôle :**
  - Sur ce que vous avez compris.
  - Et un peu de code (mais on reste flexible vis à vis de la syntaxe).



# Questions avant de commencer

---

- Avez vous déjà utilisé :
  - Docker ?
  - Node-Red ?
  - MQTT ?
  - Kafka ?
- Avez vous déjà construit des architectures client-serveur ?
- Avez vous déjà fait du javascript / nodejs ?

# Qu'est ce qu'une architecture logicielle ?

---

“  
The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those **elements**, and the **relationships** among them.”

-- Bass et al.

“  
Architecture is the fundamental **organization** of a **system** embodied in its **components**, their **relationships** to each other, and to the **environment**, and the principles guiding its design and evolution.”

-- IEEE 1471

# Une architecture logicielle ...

---

- Se concentre sur les **éléments important** (trouver le bon niveau d'abstraction).
- Doit **répondre aux besoins** utilisateurs.
- **Peut** suivre un style architectural.
- Une architecture **n'est pas figée** mais appelée à évoluer avec le temps.
  - Pour différentes raisons (e.g., passage à l'échelle, nouveaux besoins / fonctionnalités)
- Une architecture peut **influencer la manière dont une équipe s'organise**.

# Pourquoi ?

---

- Fourni une **description / un plan du travail** pour le développement et l'intégration. Devrait permettre **d'organiser le travail** dans une équipe.
- Description à partir de laquelle on peut **analyser** la conception d'un système (à différents niveaux d'abstraction).
- Faciliter la **compréhension du système** et la **communication dans une équipe**.
- Favorise **l'identification des éléments** d'un système et leur **réutilisation**.
- Facilite l'identification des points du système qui peuvent **varier, évoluer**.

# Préoccupations lorsque l'on définit une architecture

1. Doit pouvoir être **implémentée**.
2. Doit pouvoir être **déployée**.

Mais il faut aussi considérer:

- **Modularité**: séparation des préoccupations, cohésion et découplage
- **Evolutivité**
- **Performances**
- **Passage à l'échelle** et **disponibilité**
- **Fonctionnalités**
- **Sécurité** et vie privée
- **L'infrastructure** (surtout dans l'IoT)

Tous ces éléments sont liés les uns aux autres !

# Comment spécifier / décrire une architecture logicielle ?

---

- On peut très bien **commencer avec des boîtes et des flèches !**
- UML propose une **syntaxe** standard pour écrire des modèles.
  - Certains des **langages d'UML peuvent être utiliser pour spécifier des architectures logicielles.**
  - Mais en **aucun cas on est obligé d'utiliser UML.**
- Plein d'autres langages pourraient être utilisés (e.g., OASIS TOSCA)

# Viewpoints

---

- Les vues pour une architecture logicielle :
  - **Logique** (*UML : Paquetage*)
    - Décrit le système comme un ensemble de sous-système groupés logiquement.
  - **Implémentation** (*UML : Composants*)
    - Décrit l'implémentation du système sous la forme d'un ensemble de composants. Ainsi que les responsabilités de chaque composants.
  - **Déploiement** (*UML : Déploiement*)
    - Décrit l'environnement, infrastructure dans lequel le système sera déployé et les relations, dépendances entre les composants du système et l'environnement, infrastructure.



# Diagrammes de composants

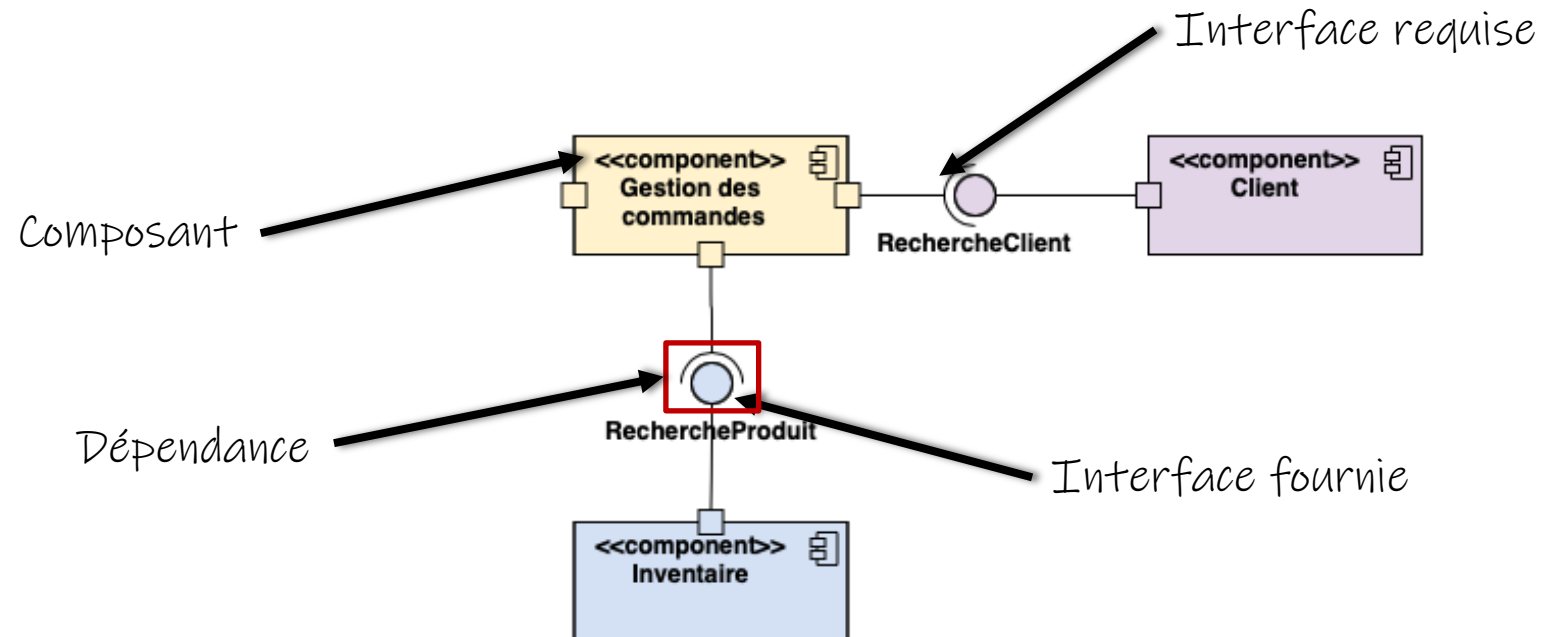
---

- Une représentation de haut niveau de l'architecture du système.
  - Décrit comment grouper des éléments d'un système dans des entités que l'on appelle **composants**.
- Permet de décrire les composants d'un système et leurs interactions. On parle **d'assemblage de composants**.

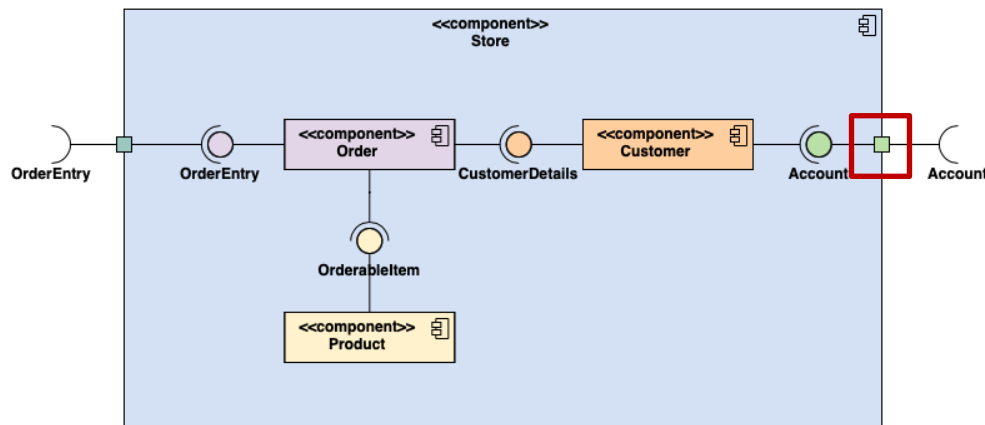


# Composants

- Un composant est une unité modulaire et autonome avec des interfaces définies.
  - Possède des **interfaces fournies**.
  - Possède des **interfaces requises**.
  - Souvent ce qui compose un composant est caché et inaccessible.



# Composants composites et ports

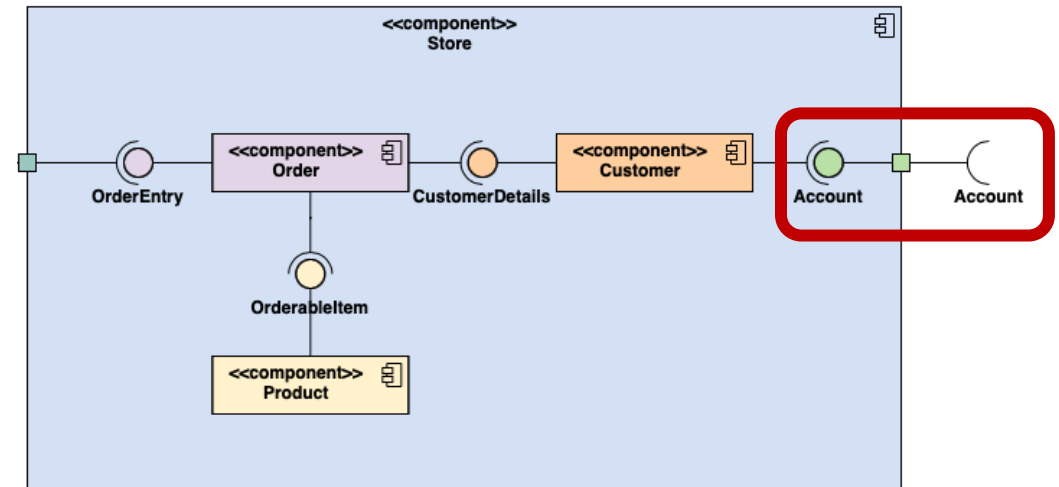


- **Ports :**

- Les ports peuvent avoir un nom. La combinaison <nom composant + nom port> doit être unique.
- Peuvent représenter des communications unidirectionnelles ou bi-directionnelles.

# Composants composites - Délégation

- **Délégation :**
  - Transmission d'information entre le conteneur et un composant interne
  - Forcement entre interfaces et ports du même genre



# Processus (un exemple)

---

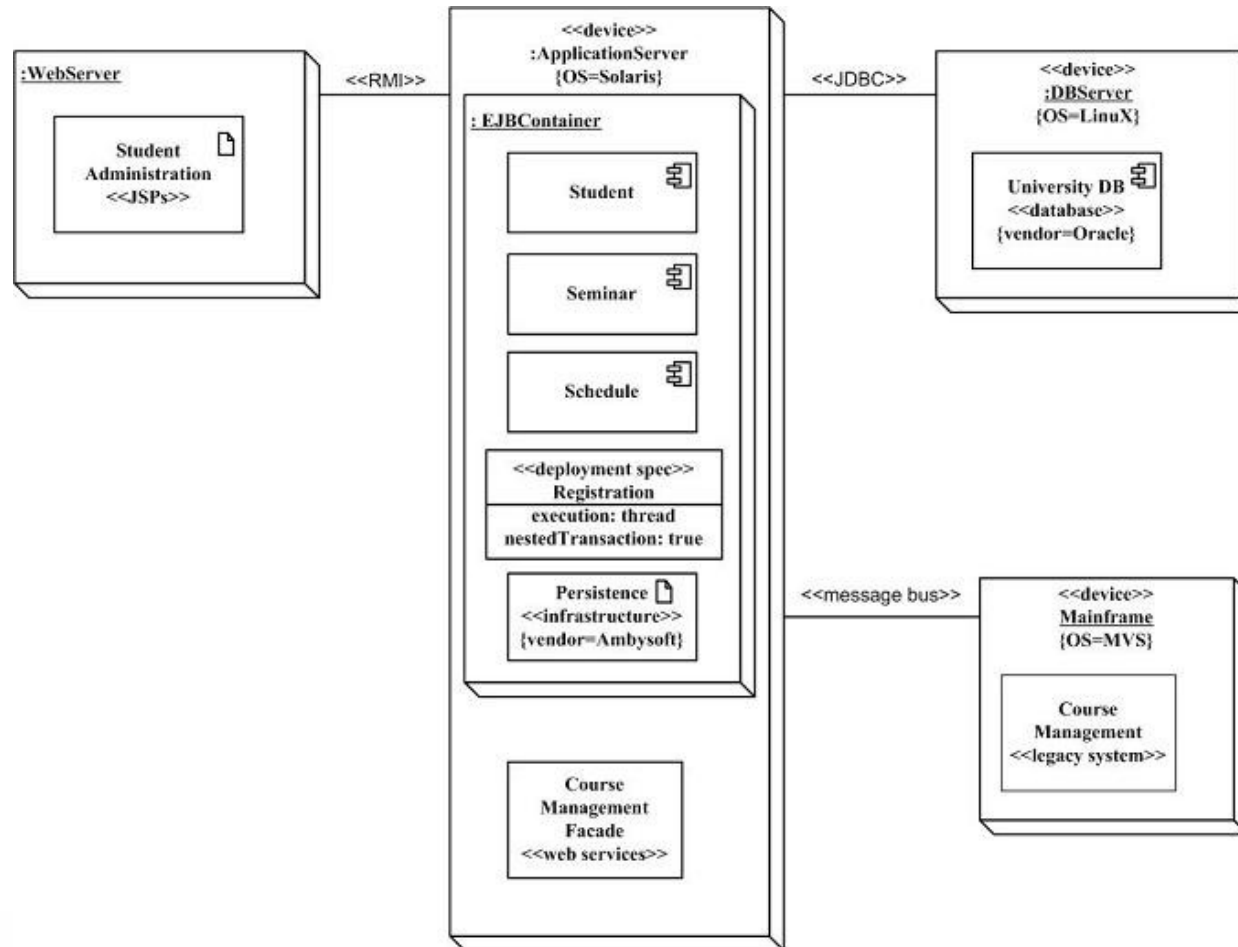
1. Identifier les modules
2. Définir leurs responsabilités
3. Définir les interfaces
4. Réaliser les connections
5. Essayer plusieurs scénarios
6. Analyser les caractéristiques de l'architecture

# Modèle de déploiement

---

- Décrit une vision statique et déclarative du déploiement du système.
- Etabli une relation entre:
  - **Artefacts** (aka. deployable artefact) : un élément physique produit par le développement logiciel, ou utilisé par lui, qui sera déployé sur un nœud (e.g., binaire, fichiers).
  - **Nœuds** : des ressources matérielles ou des environnements d'exécution.
- Expliciter les associations entre les nœuds.
- Les artefacts peuvent être reliés entre eux par des **dépendances**.

# Modèle de déploiement



# Processus (un exemple)

---

1. Décrire l'environnement / infrastructure
2. (Si déjà connu décrire les contraintes hardware)
3. Allouer les composants / modules sur les dispositifs
4. (Si hardware pas « subi » décrire les contraintes hardware)
5. Analyser les caractéristiques de l'architecture
6. Evaluer

# Infrastructures des systèmes IoT

---

Attention ! Dans les systèmes IoT les infrastructures ont des caractéristiques très différentes ! On ne peut pas toutes les approcher de la même manière !



# Le "Things to Cloud" Continuum

## CLOUD LAYER

Big Data Processing  
Business Logic  
Data Warehousing

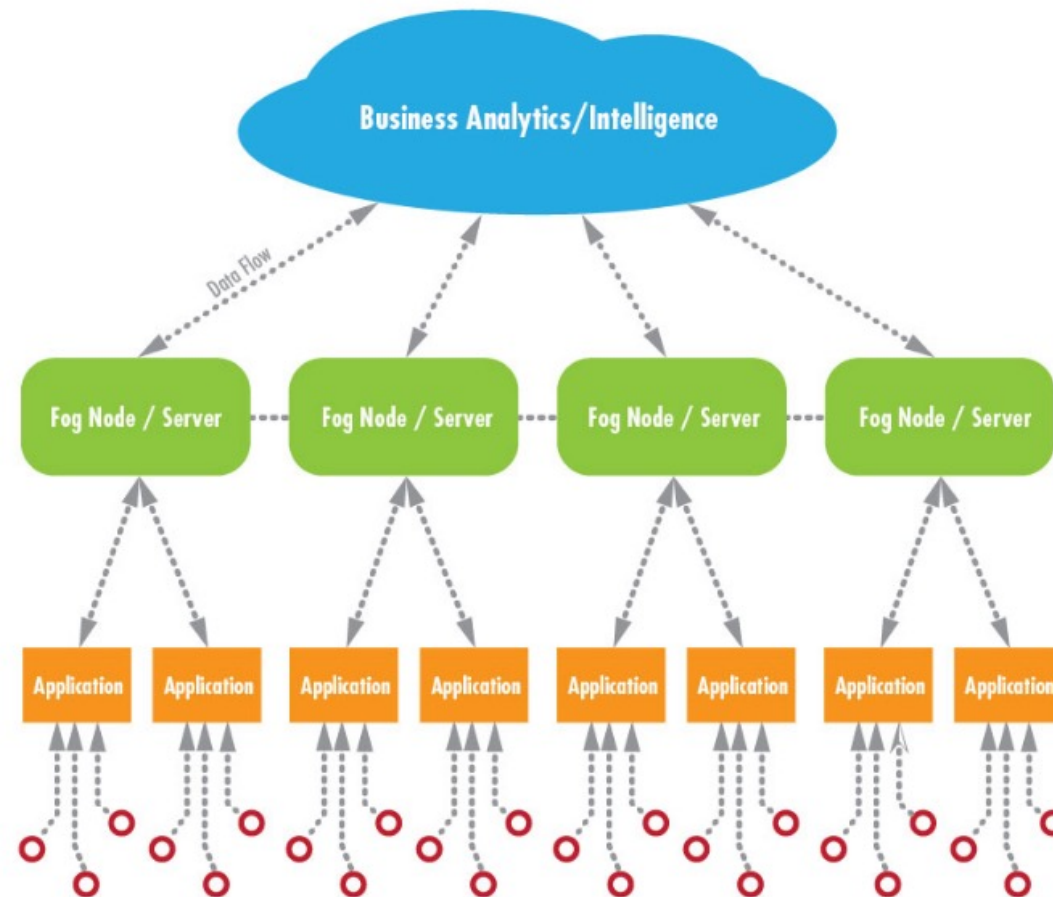
## FOG LAYER

Local Network  
Data Analysis & Reduction  
Control Response  
Virtualization/Standardization

## EDGE LAYER

Large Volume Real-time Data Processing  
At Source/On Premises Data Visualization  
Industrial PCs  
Embedded Systems  
Gateways  
Micro Data Storage

Sensors & Controllers (data origination)



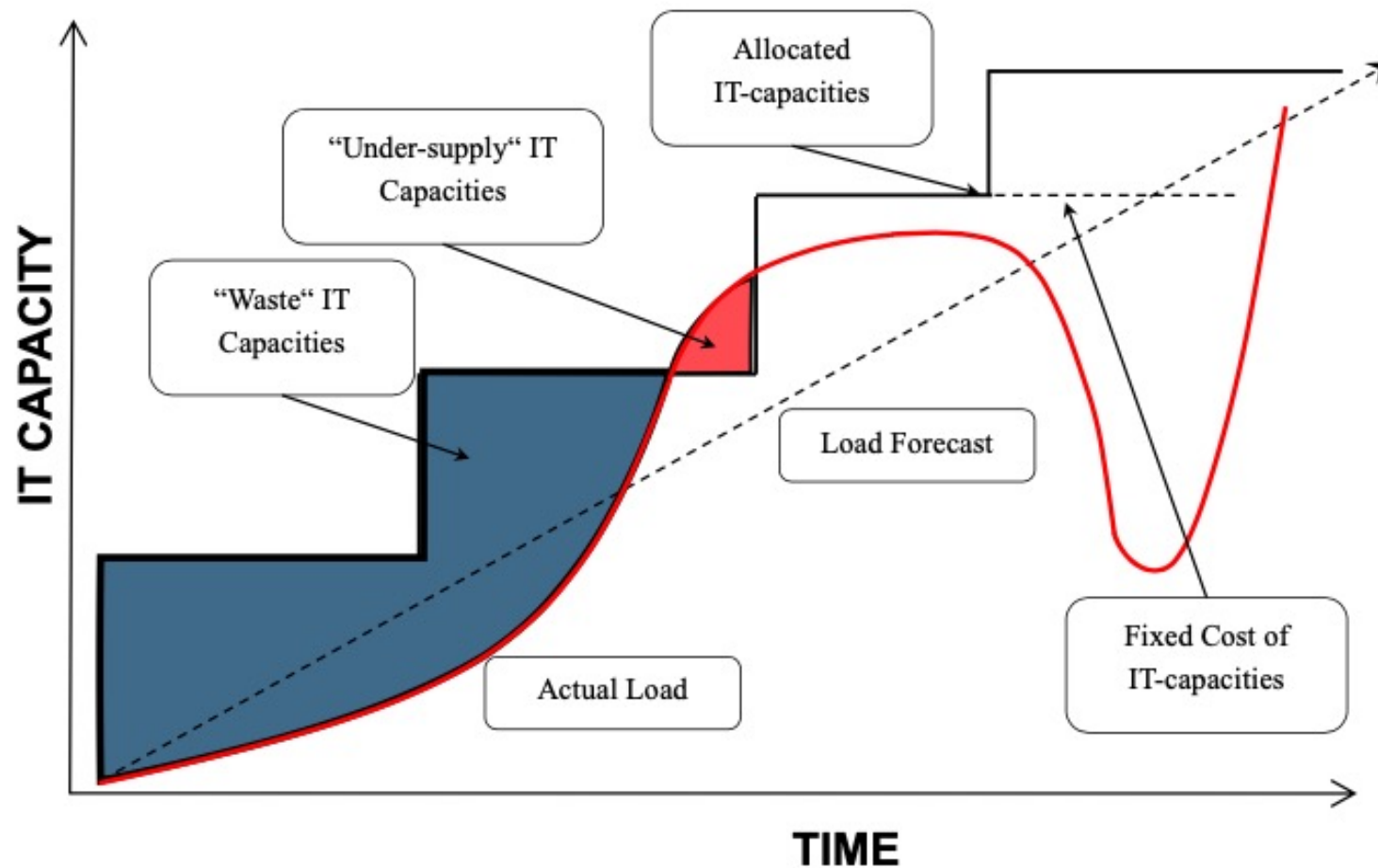


# Cloud computing

---

- Cloud computing in short:
  - **Large-scale** and **accessible on demand resources**
    - Network
    - Storage
    - Compute
    - Software
  - **Available** via Web service calls **through** the **Internet**
  - Short- or long-term access on **a pay per use** basis

# Optimize IT Capacity to the load



Courtesy of Microsoft

# Elasticity and Scalability

---

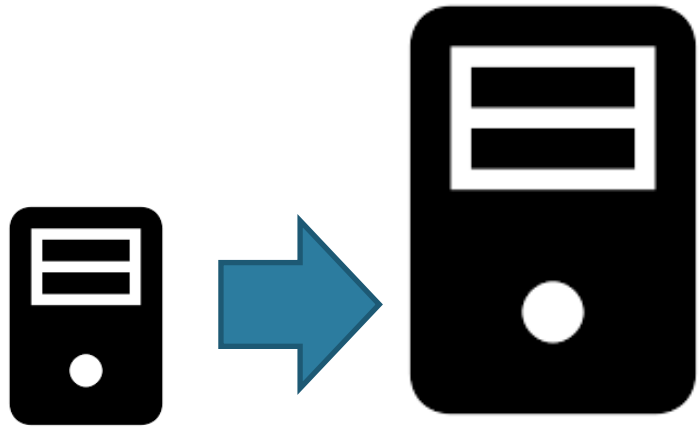
- **Scalability:** the ability of a service to sustain variable workload while fulfilling quality of service (QoS) requirements, possibly by consuming a variable amount of underlying resources.
- **Elasticity:** the ability of a service to rapidly provision and deprovision underlying resources on the fly.

**One does not guarantee the other!**

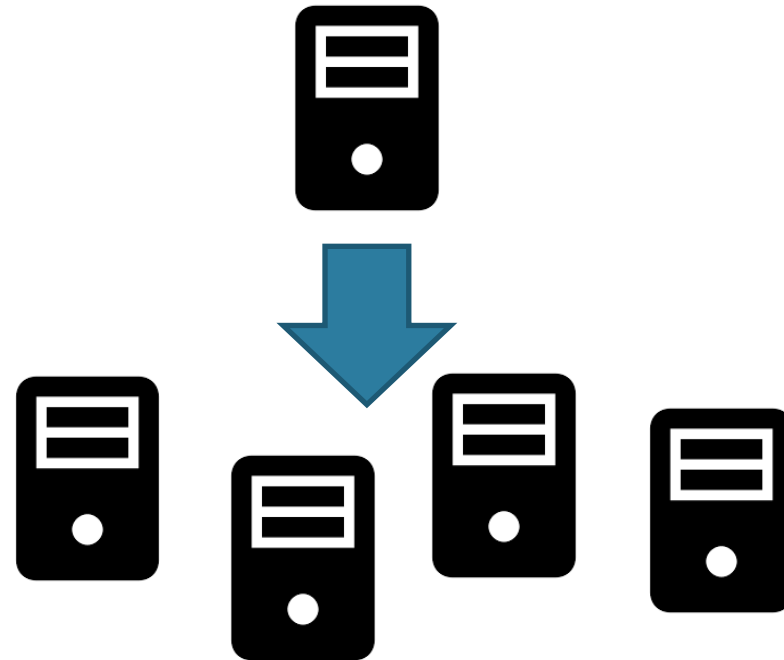
# Deux types de scalabilité

---

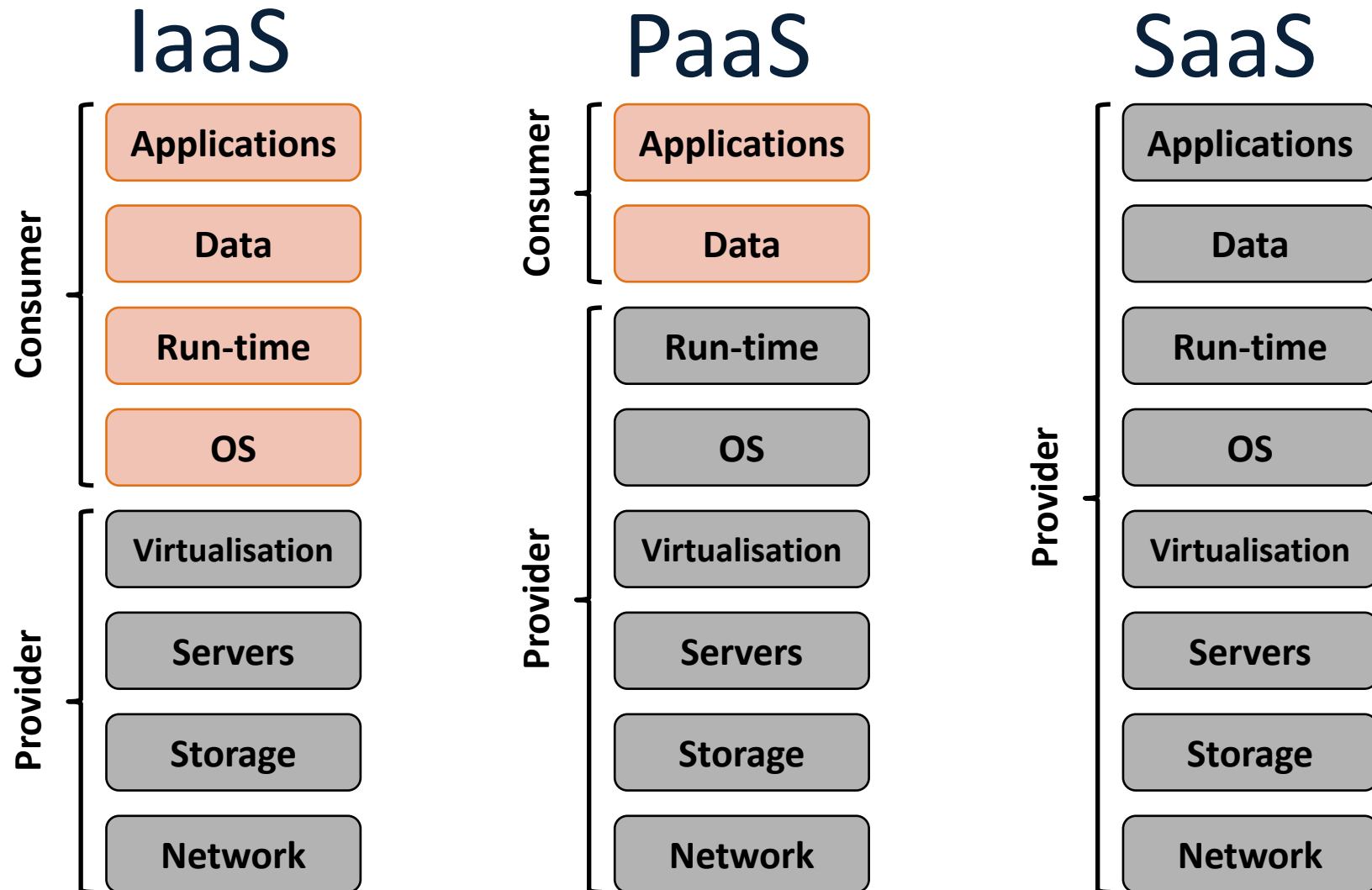
- Verticale



- Horizontale



# The Cloud computing stack



# Deployment model

---

- **Private Cloud**

- Owned by the organization. Said to be more secure as the storage and processing stays under the organization control.
- E.g., OpenStack, Cloud Foundry

- **Public Cloud**

- Hosted at the provider premises, who is in charge of its maintenance and management.
- E.g., AWS, Azure

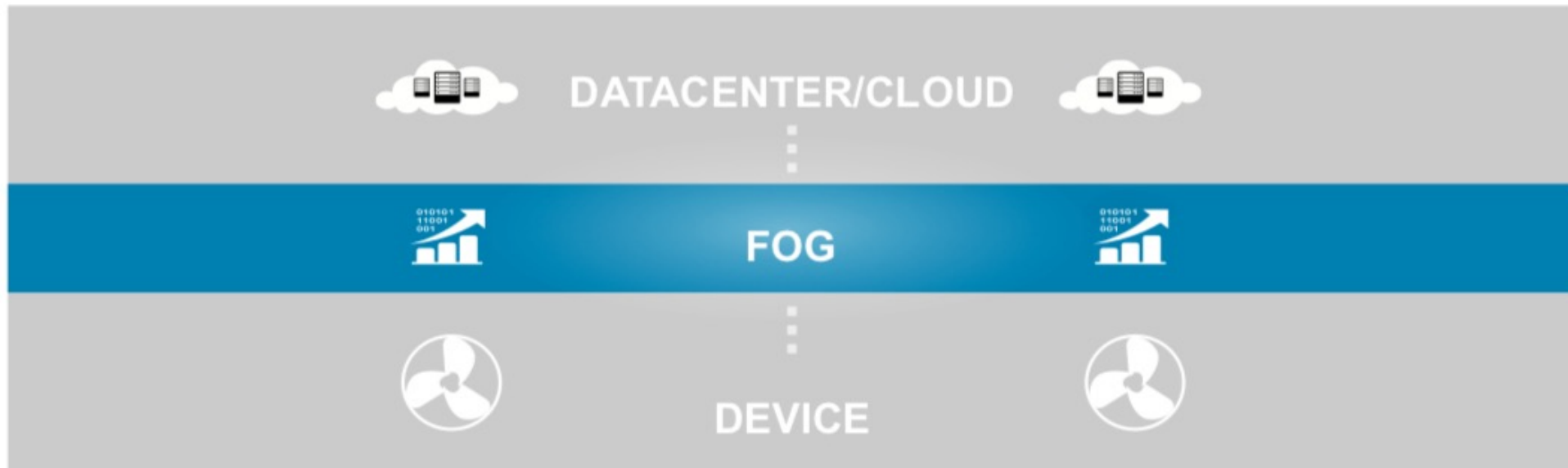
- **Hybrid Cloud**

- Composition of two or more public and private Clouds



# Fog Computing

---



# Why fog computing?

---

- Location aware
- Geographical distribution
- Mobility
- Large number of nodes
- Low latency

## When to Consider Fog Computing

- Data is collected at the extreme edge: vehicles, ships, factory floors, roadways, railways, etc.
- Thousands or millions of things across a large geographic area are generating data.
- It is necessary to analyze and act on the data in less than a second.

--source: cisco

# Edge & Fog

---

- La nuance est mince, et il n'y a pas vraiment de consensus sur la différence entre les deux.
- Globalement les objectifs sont les mêmes.
- Certains considèrent que les dispositifs Edge sont plus proche des sources de données et actionneurs.

# Edge VS Cloud

---

## Cloud infrastructure

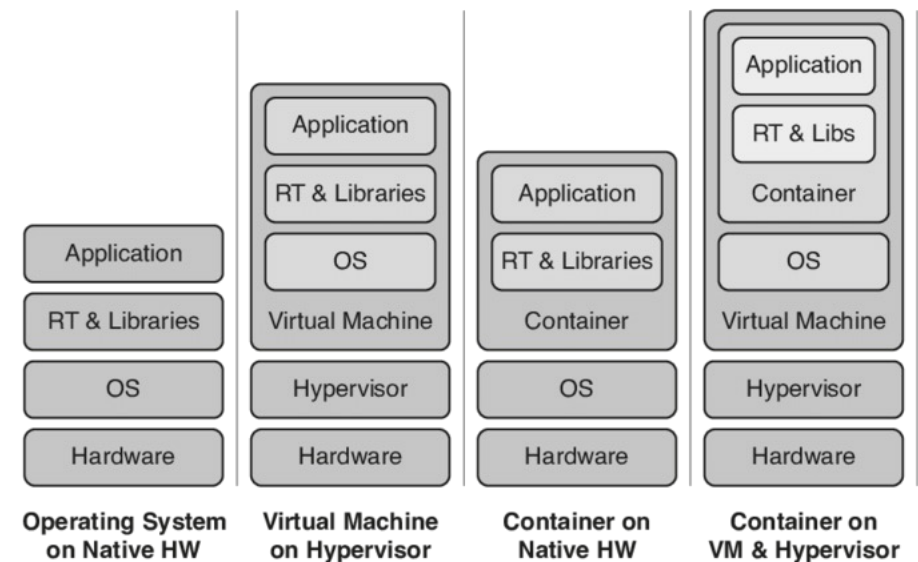
- “Centralized”
- large grain geographical distribution
- Homogeneous
- Infinite resources on demand
- 99.9 availability

## Edge infrastructure

- Decentralized
- Fine grained geographical distribution
- Heterogeneous
- Limited resources “suffered”
- Unknown availability

# Hétérogénéité

- No single layer of abstraction due to diversity in:
  - Computing power
  - Storage
  - Network
  - Cannot necessarily run virtualization techniques
  - Cannot necessarily run same OS and thus different libraries



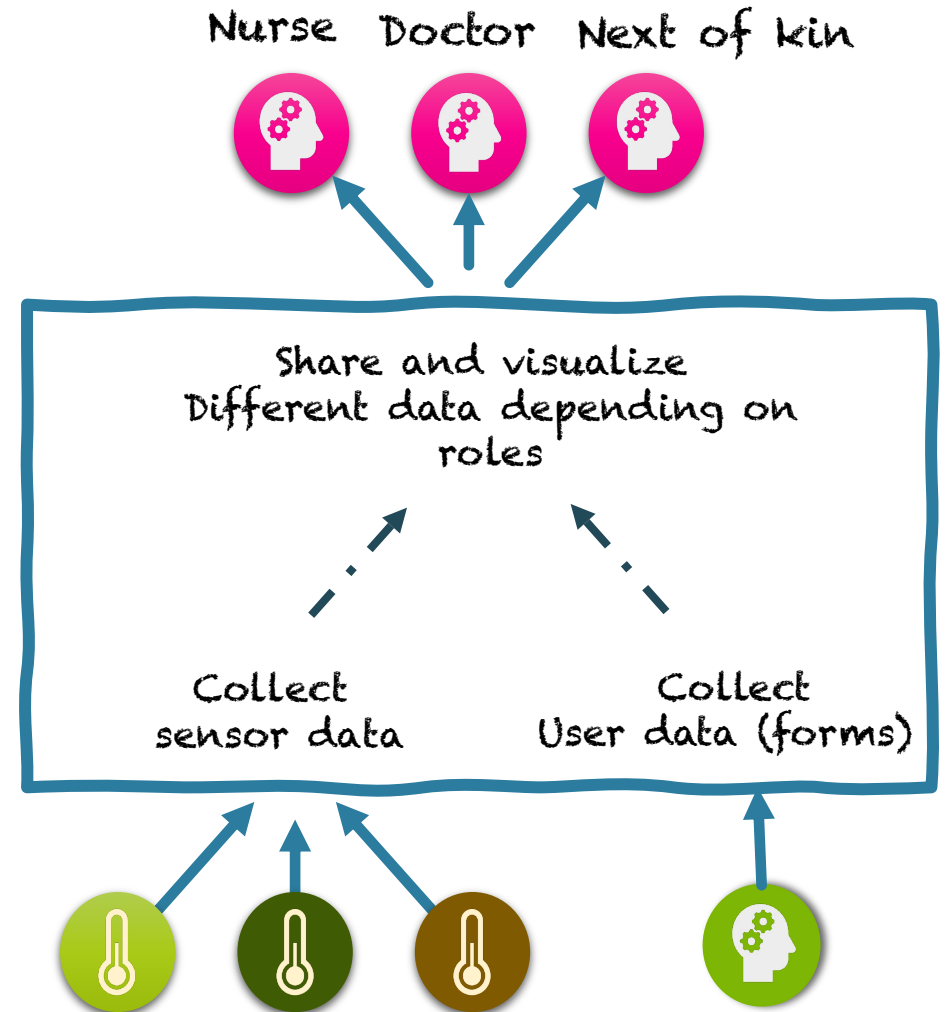
# IoT – Devices spaces !

---

- Dispositifs très **hétérogènes**.
- Sur lesquels il est parfois possible de **déployer** du code.
  - Généralement pas assez puissant pour supporter de la virtualisation.
  - Donc déploiement lent et long.
- Pas toujours **directement** connecté à Internet.
- Pas toujours **disponible**.
  - Plus de batteries
  - Plus de réseau
  - ...
- Dans un **contexte** cyber-physique qui leur est propre

# Exercice

- **Contexte : Une solution d'e-santé.**
  - Aider les personnes âgées à rester à la maison.
  - Partager des informations avec certaines personnes.
- Il faudrait mais pour ce cours **on ne se préoccupe pas :**
  - De la sécurité / vie privée.
  - Business model.



# Exercice

---

- En groupe de 2.
- Proposez une architecture du système de e-santé présenté précédemment
  - Utilisez des diagrammes UML (Vous pouvez commencer par des flèches!).



# Exercice 2

---

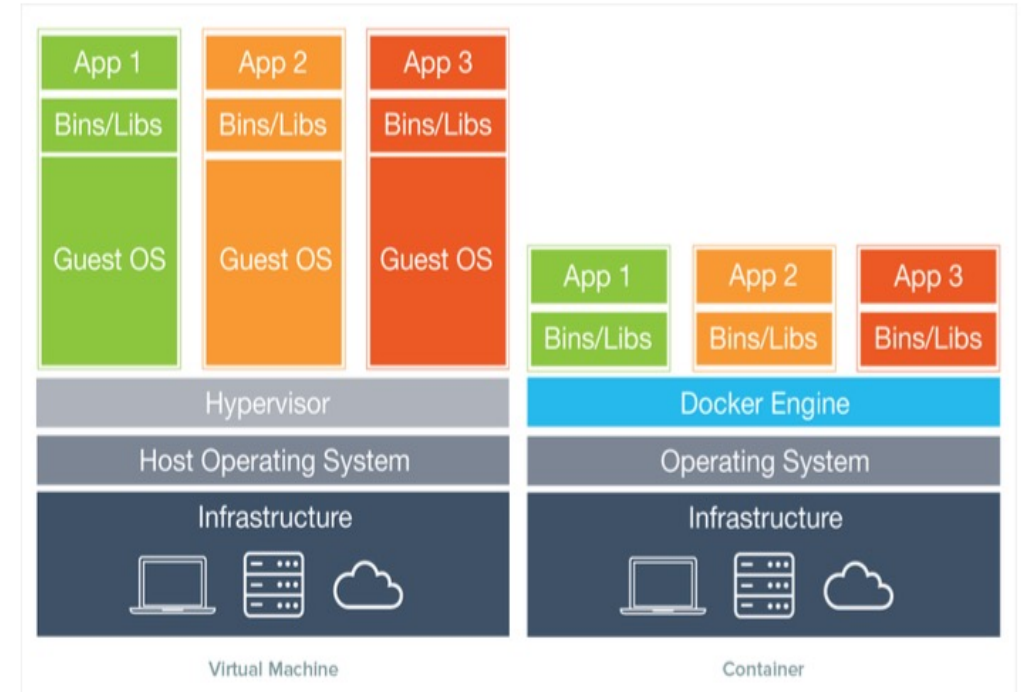
- Sonnette, serrure, et interphone intelligente.
  - Le dispositif est équipé d'une caméra.
  - Ouverture de la porte si la caméra reconnaît la personne.
    - Attention on doit toujours pouvoir rentrer.
  - Allume les lampes la nuit quand quelqu'un se présente devant la sonnette.
    - Pas si c'est un animal!
  - Si personne ne répond à l'interphone, une communication est établie avec le smartphone du propriétaire.
- **En groupe de 2, présentez une architecture pour un tel système.**

# Docker

On survol docker car nous l'utiliserons dans plusieurs TPs !

# C'est quoi

- Une mécanisme de containerisation et de gestion de container :
  - Virtualisation **légère** avec des images **immutable** et légères déployable très **rapidement**.
- **?!!**
- Docker permet d'exécuter vos applications dans des containers isolés et connectés via des réseaux virtuels.
  - Les containers sont réutilisables et portables.
  - Le cycle de vie des containers peut être géré dynamiquement.



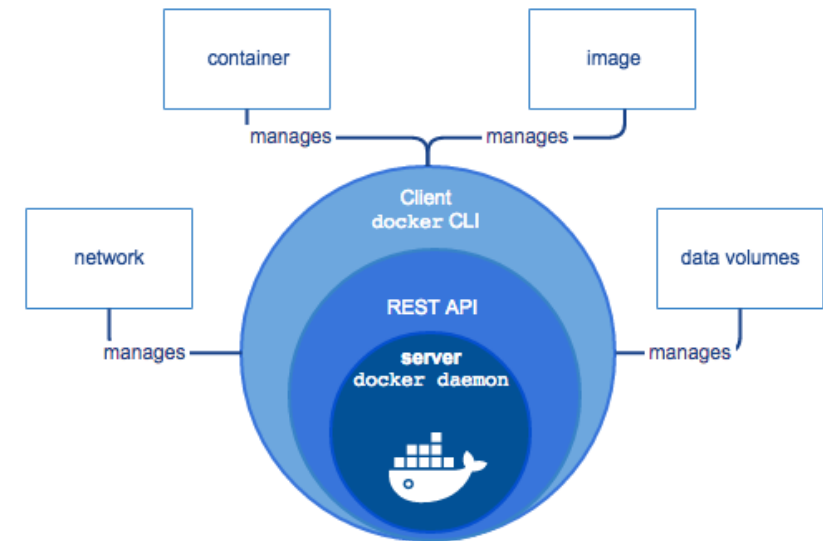
# Concepts

---

- **Image** : un instantané immuable du système de fichier (donc qui inclus l'application et ses « dépendances »).
- **Container**:
  - Un système de fichier temporaire:
    - Construit sur le système immuable de l'image
    - Détruit lorsque le container est tué sauf si on crée une image avant cela
    - Accessible en lecture et écriture.
  - Un groupe de processus.
  - Une pile réseau (172.17.x.x)

# C'est aussi

- Un moteur d'exécution pour gérer le cycle de vie des containers et construire des images
- Des APIs:
  - Pour gérer, publier des images
  - Construire des images
  - Gérer des containers
  - Composer des containers
  - Passer à l'échelle (gérer des cluster d'images)



# Bénéfices

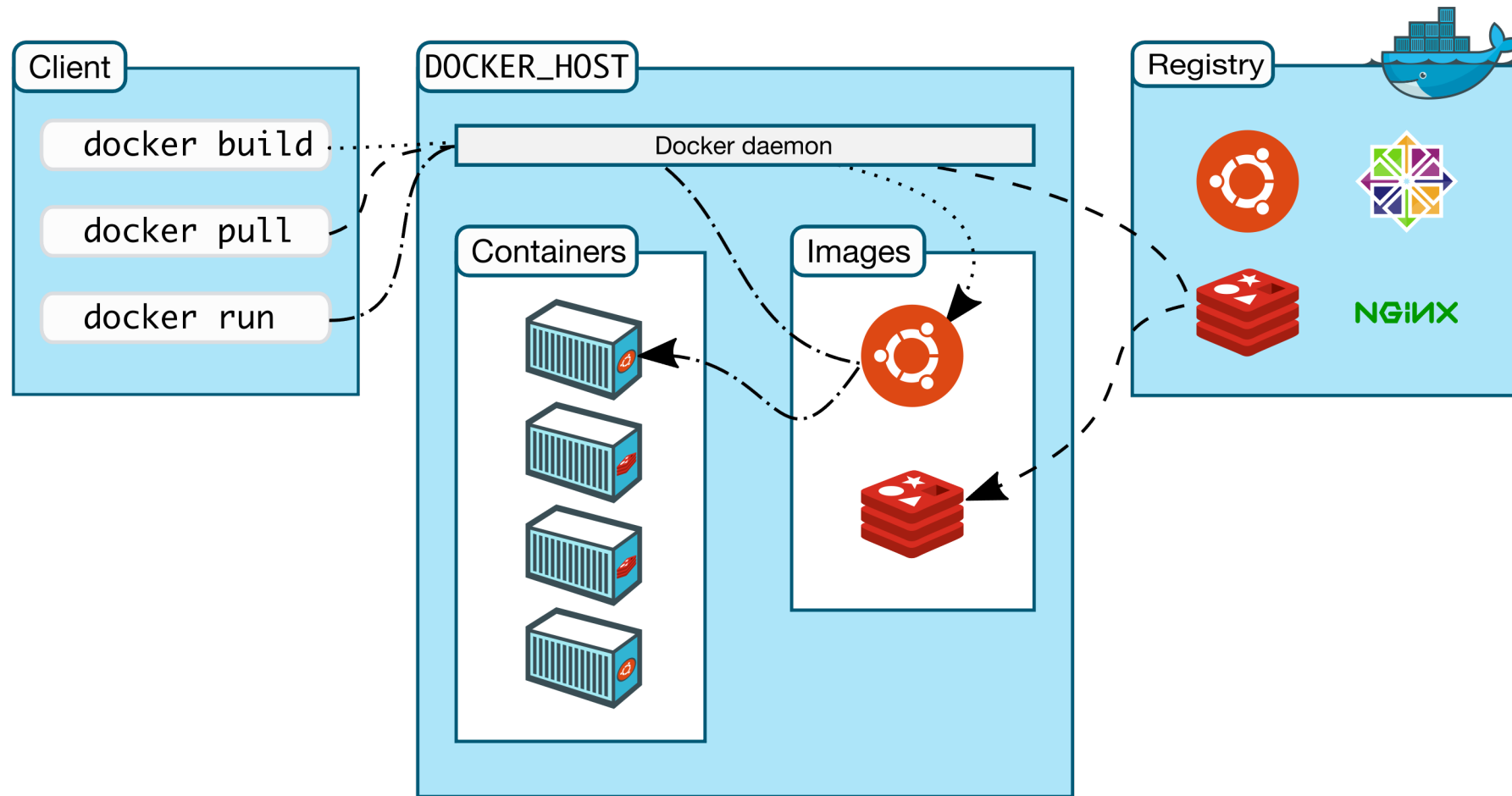
---

- **Approche grey-box** - on construit un déploiement fonctionnel et immutable pour une version du logiciel, que l'on peut ainsi (re)déployer :
  - Automatiquement et rapidement sans avoir à reconstruire toute la pile logicielle.
  - Indépendamment de la plateforme sous-jacente (à l'exception de l'architecture ARM, x86, même si ...).
  - A tout instant, puisque l'image est immutable, si elle est fonctionnelle, elle peut toujours être relancée.
  - Et facilement partager.
  - Et réutilisée pour construire d'autres images.

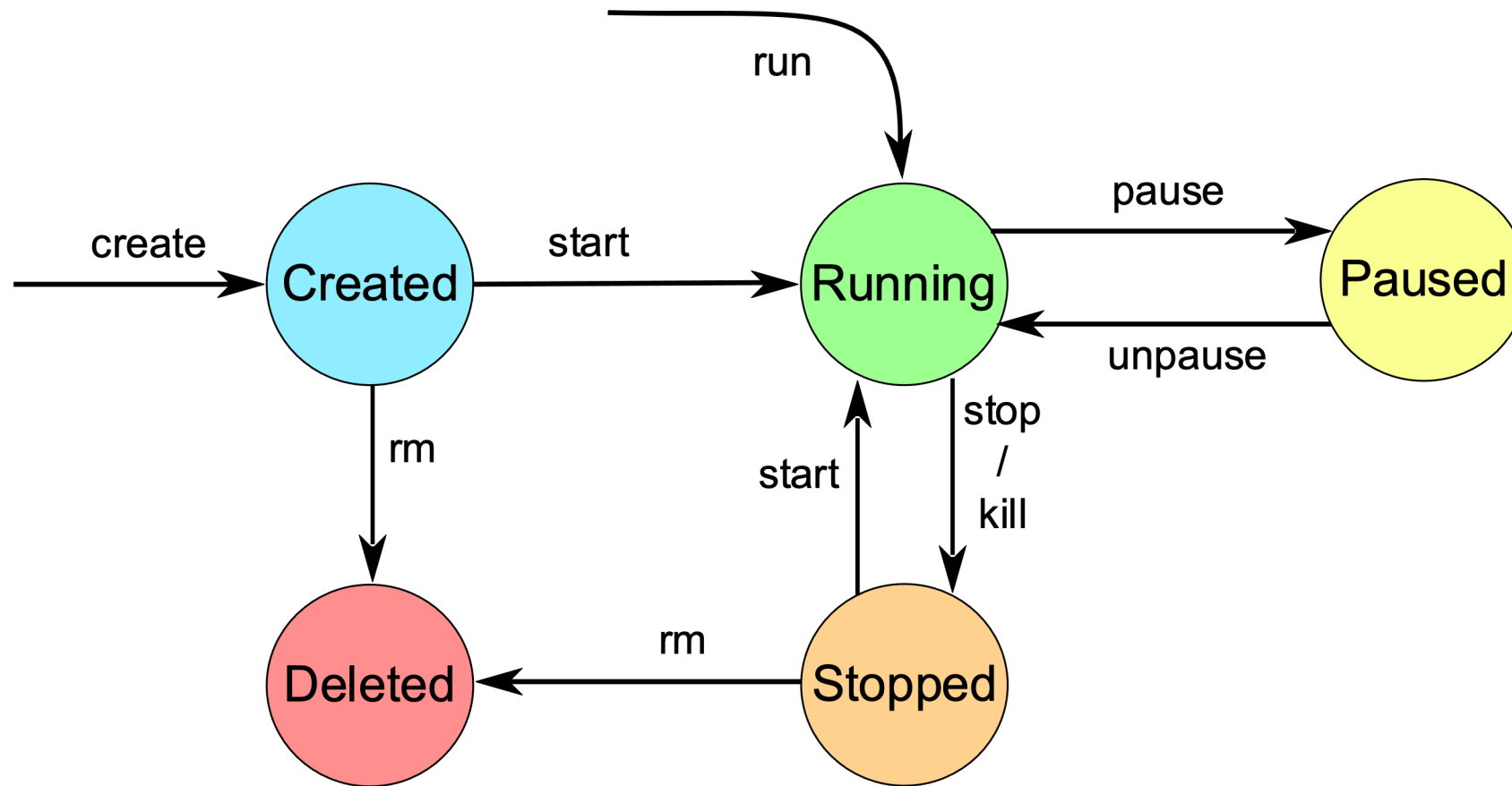
## **MAIS :**

- 1. On ne sais pas toujours ce qu'il y a dans une image !**
- 2. Une sorte de vendor lock-in !**

# Architecture



# Cycle de vie d'un container





# Quelques commandes

- Lister les containers existant

```
docker ps -a
```

- Créer un container a partir d'une image

```
docker create <image>
```

← Système de fichier initialisé  
à partir de l'image

- Créé (si pas déjà fait) et démarre un container

```
docker run <image>
```

← -d detached mode

- Démarrer un container

```
docker start <container>
```

← -v /hostpath:/ctrpath

- Arrêter un container

```
docker stop <container>
```

- Tuer un container

```
docker kill <container>
```

# Quelques commandes

---

- Supprimer un container  
`docker rm <container>`
- Créer une image a partir d'un container  
`docker commit <container> <image>`
- Exécuter une commande dans le container  
`docker exec <container> args`
- Accéder aux logs d'un container  
`docker logs -f <container>`

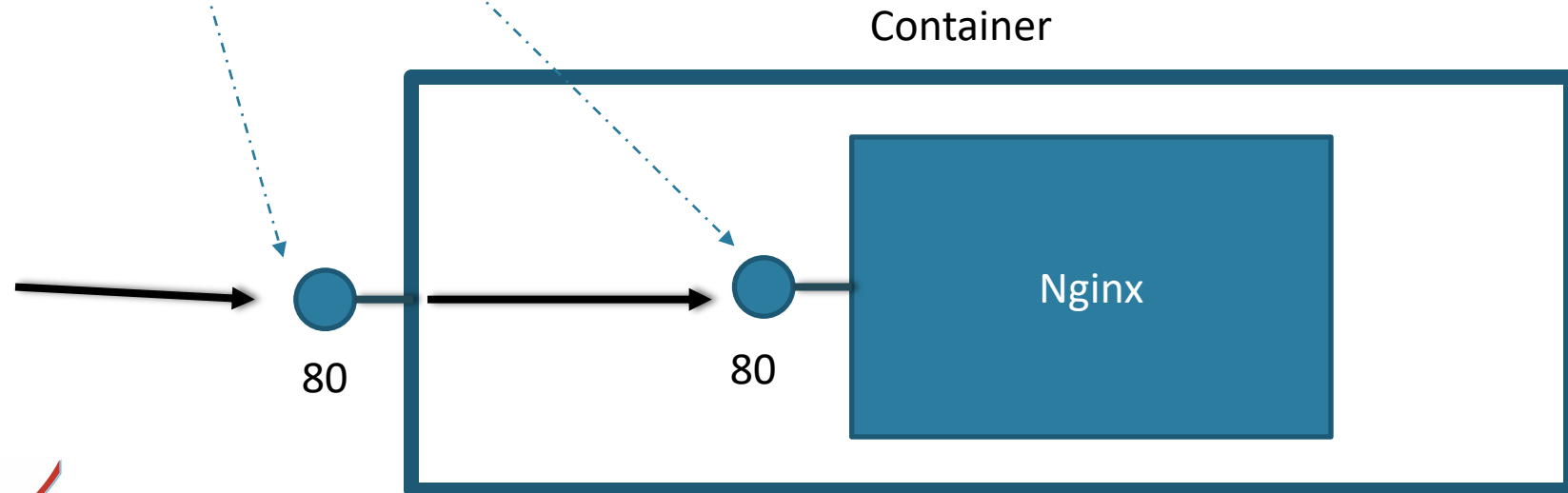
← Système de fichier terminé  
(persistent entre start et stop)



# Gestion des ports

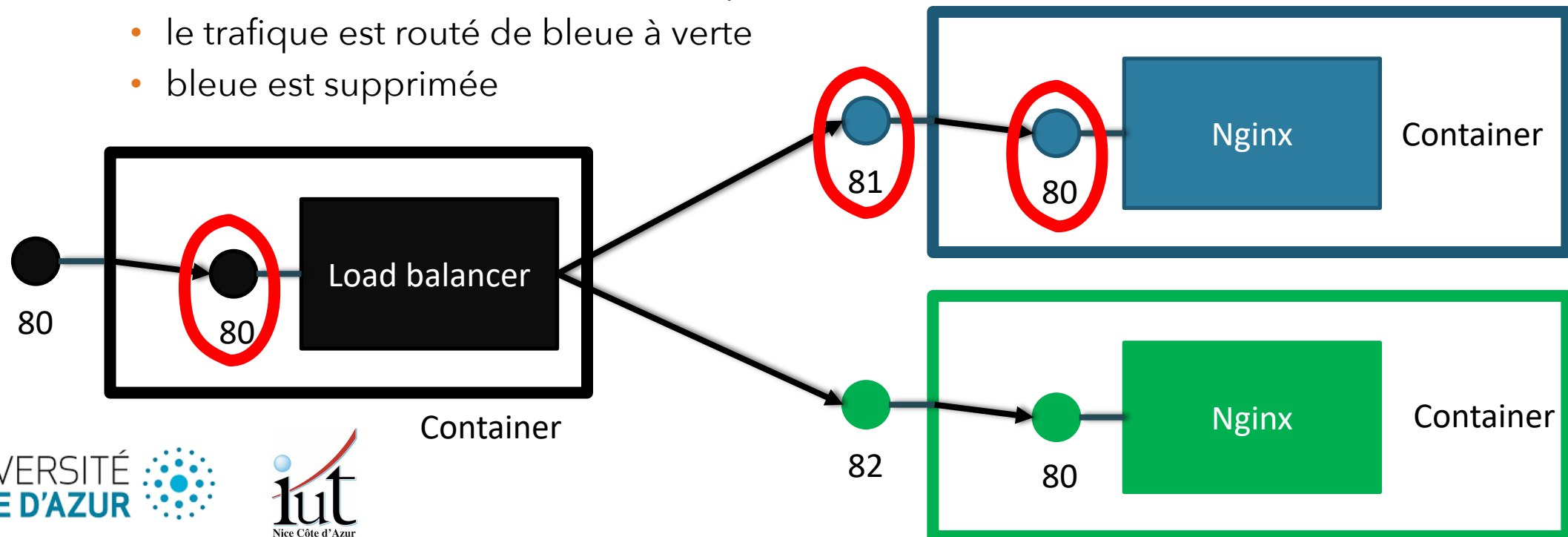
- Par défaut, les containers ne sont pas accessible depuis l'exterieur
- Mais il est possible d'exposer des ports

```
docker run -p 80:80 nginx
```



# Gestion des ports

- L'exemple du déploiement bleu/vert :
  - Une version bleu est en production
  - Une nouvelle version verte est disponible
  - Elle est déployée en parallèle de bleu
  - Une fois verte considérée comme prête:
    - le trafic est routé de bleue à verte
    - bleue est supprimée



# Creation d'images

---

## 1. Commit d'un container.

## 2. Docker builder:

1. Un langage pour décrire comment construire une image.  
<https://docs.docker.com/reference/builder/>
2. Un mécanisme pour construire l'image comme une succession couches.  
(avec un cache pour les layers n'ayant pas changés)  
`docker build -t <tag> <Dockerfile>`
3. Prend en entrée un fichier Dockerfile et à accès aux fichiers dans le répertoire ou se situe le fichier Dockerfile.

# Dockerfile

---

# Couche 1 = A partir de cette image d'ubuntu

```
FROM ubuntu:18.04
```

# Couche 2 = On exporte une variable d'environnement

```
ENV APP_HOST='192.168.1.23'
```

# Couche 3 = On copie le répertoire app qui se trouve au même emplacement que Dockerfile

```
COPY . /app
```

# Couche 4 = On exécute une commande pour construire l'application

```
RUN make /app
```

# Couche 5 = Commande à exécuter dans le container

```
CMD python /app/app.py
```

# Exercice

---

- Au moins les sections 1, 2, et 3 de l'exercice « Docker » sur l'ENT.

# Styles architecturaux

1. Pipeline.
2. Multi-couche.
3. Event-Driven Architecture.
4. Service-Oriented Architecture.



# Rappel : Styles architecturaux

---

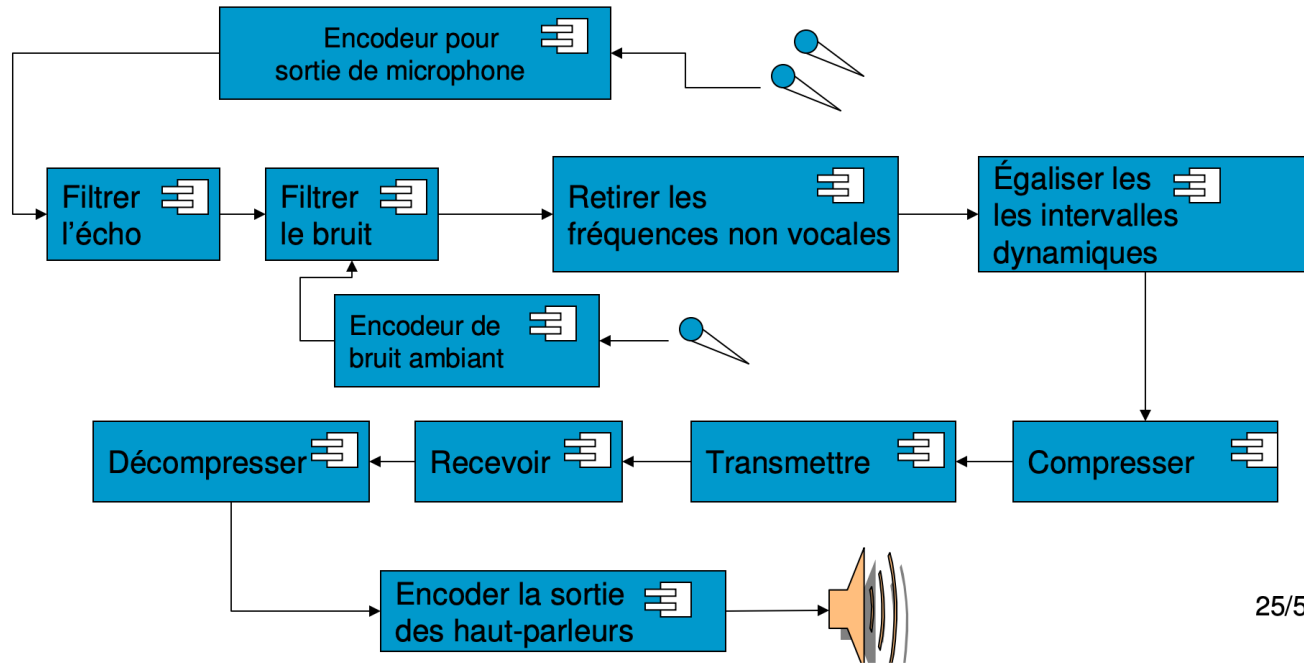
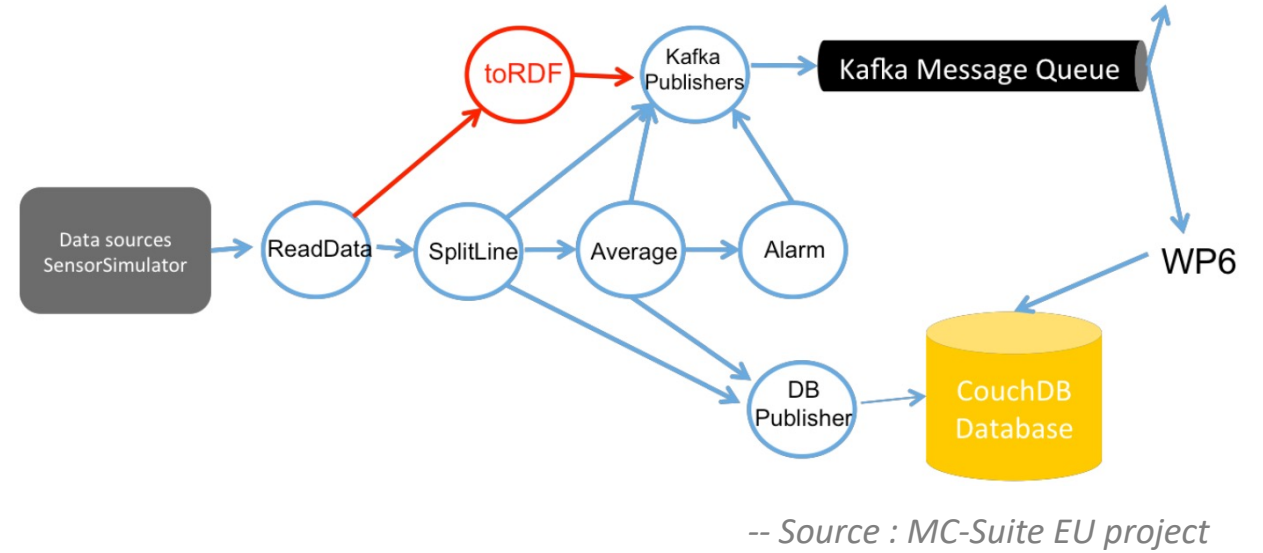
- Un pattern / **patron** décrivant un type d'architecture **adapté à une situation / à un contexte**.
  - Modèle **éprouvé** et enrichi avec le temps par **l'expérience** et les retours de développeurs.
  - Définit typiquement :
    - Les composants et interactions entre les composants (types, règles et topologie).
    - Des exemples de systèmes construits selon le patron.
    - Spécification des objectifs et du comportement du patron.
- Il existe de nombreux styles architecturaux !

# Architecture en pipeline

---

- Architectures pour lesquelles on enchaîne (sous la forme d'un pipeline) des traitements les uns à la suite des autres.
- Convient bien aux systèmes de traitement de (flux) données.
  - E.g., traitement vidéo/son, traitement de flux de données de capteurs.
- On y retrouve les concepts suivants :
  - **Sources de données (aka. Data source)** : le composant qui produit les données.
  - **Filtres (aka. Data processing components)** : composant réalisant un traitement indépendant. Consomme une ou plusieurs données et produit une/des données.
  - **Canal (aka. Data stream)** : à travers lequel circulent les données, typiquement unidirectionnel.
  - **Pipeline (aka. Data flow)** : l'orchestration des canaux et filtres.

# Exemples



25/51

-- Source : Cours de conception architectural de Yann-Gaël Guéhéneuc

# Streams / flux

---

- **Efficacité en espace :**

- Pas de besoin de mettre en buffer toutes les données.
- *E.g., traiter un fichier d'un To ? Pas toujours possible avec un buffer.*

- **Efficacité en temps :**

- Pas besoin d'attendre que toutes les données soient dans le buffer, on les traite au fur et à mesure.
- *E.g., pour compresser un fichier, pas besoin d'attendre que le fichier soit tout entier dans le buffer.*

# Stream processing

---

- De nombreux frameworks de stream processing ont adoptés cette architecture pour lesquels canaux, filtre, etc. sont devenus des concepts clés du langage :

- Apache Storm.
- Apache Heron.
- Apache Flink.



- Les pipeline JavaScript de Node.js sont aussi un moyen de mettre en place de telle architectures !



# Pro and Cons

---

## Avantages

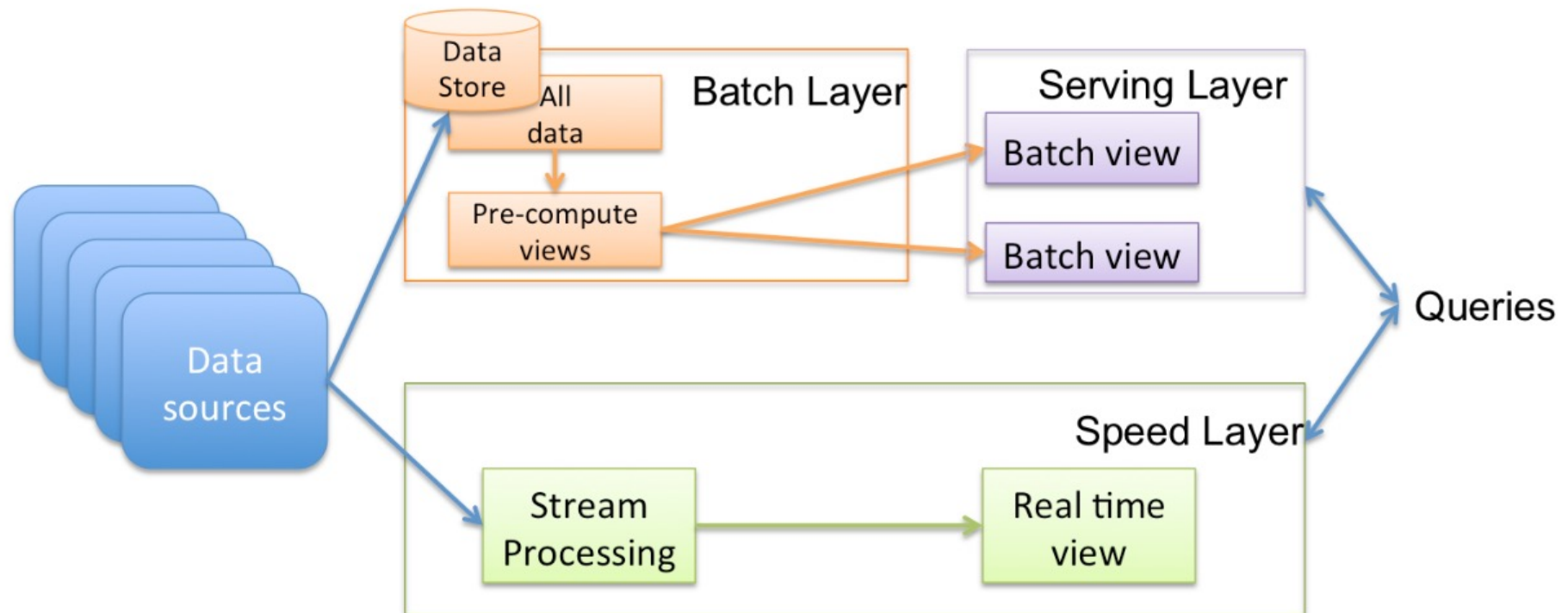
- Efficace pour le traitement de flux de données (cf. slide précédent).
- Très flexible, facile de faire évoluer le pipeline.
  - Bon découplage et bonne réutilisabilité des filtres.

## Inconvénients

- Pas vraiment adaptés à d'autres contextes que le traitement de données.

# Lambda architecture

- Combine typiquement une architecture en pipeline avec un/des autres styles d'architectures.



# Exercice

---

- TP2 :
  - Création d'une application reposant sur une architecture de type pipeline.
    - On réfléchit à l'architecture (on la dessine).
    - On implémente l'application.
- Quelques « take away » de ce TP :
  - On a un pipeline.
  - Grâce à une bonne séparation des préoccupations : on peut réutiliser des composants qui existent déjà, on peut facilement faire évoluer notre application.
  - On travaille sur des flux avec des messages : couplage faible entre les composants



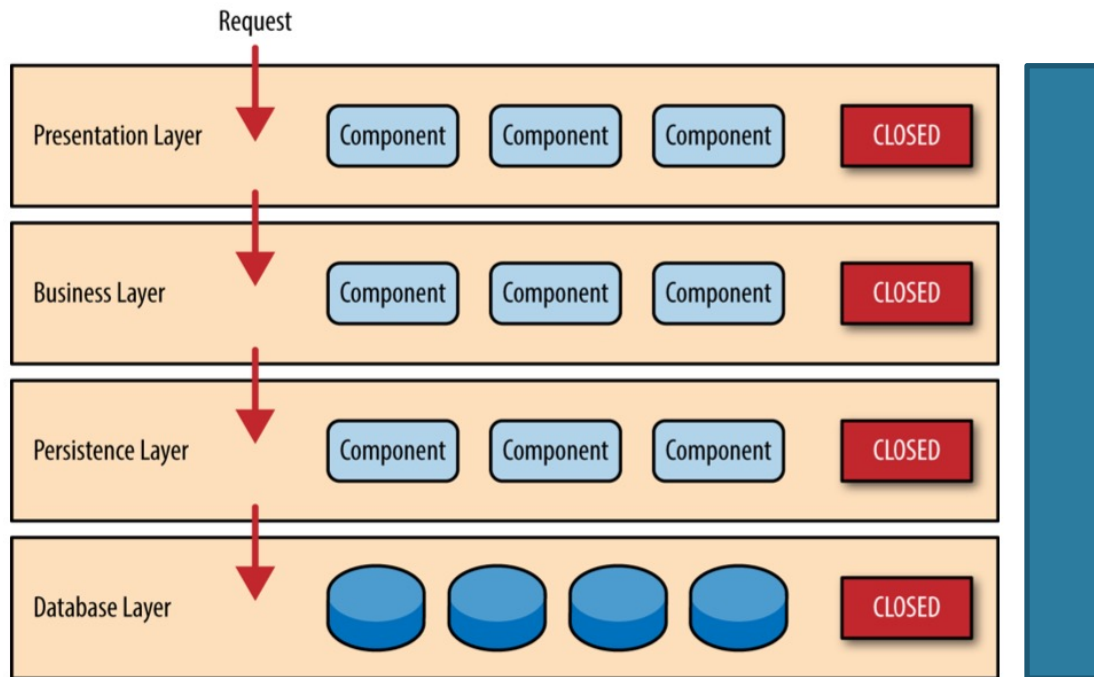
# Architectures multi-couches

- Probablement le type d'architecture le plus commun. Aussi connu sous le nom d'architecture n-tiers.
  - (Parfois on fait le distinguo entre n-tiers et multi-couches en considérant que les architectures n-tiers sont distribués.)
- **Les composants dans l'architecture sont organisés dans des couches.** Les couches sont **organisées hiérarchiquement** :
  - Chaque couche à un rôle spécifique.
  - Le nombre de couches n'est pas spécifié mais dans les faits on retombe souvent sur 3-4 couches :
    - Presentation
    - Business
    - Persistence
    - Database

} *Parfois groupés ensemble*



# Architecture multi-couches



- Une couche peut englober plusieurs composants.
- **Séparation des préoccupations :** Chaque couche forme une abstraction et se concentre sur son objectif :
  - Les autres couches n'ont pas besoin de connaître des détails de comment.
  - Scope des composants dans une couche clairement défini.
- **Isolation :** Une couche n'interagit qu'avec les autres couches qui l'entourent. Parfois on ouvre une couche
  - Meilleure évolutivité (changer le layer Persistence ne change pas le layer Presentation)

# Architectures multi-couches

---

- Un système complexe est construit en superposant des couches.
- En ce qui concerne l'isolation des couches, il peut y avoir deux approches:
  - **Système fermé** : une couche n'a accès qu'aux couches adjacentes.
  - **Système ouvert** : toutes les couches ont accès à toutes les couches. Attention, dans ce cas il ne faut pas faire n'importe quoi au risque de faire disparaître la notion de couches !

# Préoccupations transverses

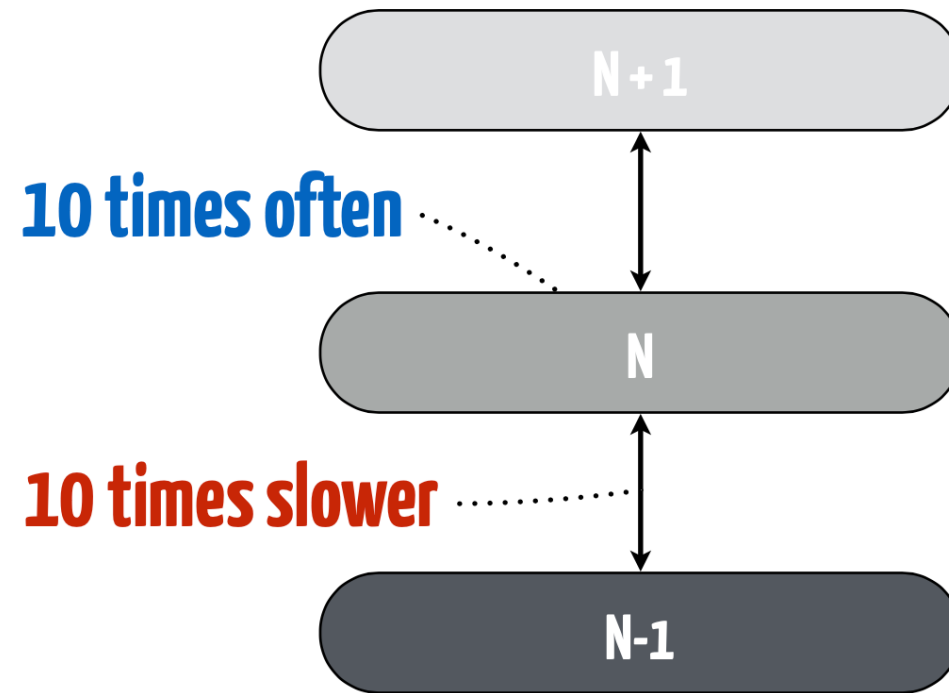
---

- Il existe des préoccupations qui sont transverses à toutes (ou plusieurs) les couches - i.e., qui doivent être prises en compte dans toutes les couches :
  - **Sécurité** : Authentification, intégrité, non-repudiation...
  - **Services techniques** : logs, traces, monitoring ...

# Warning !

---

- Sinkhole anti-pattern:
  - Faire des couches qui ne font rien sauf faire suivre des requêtes !
- The rule of 10 !

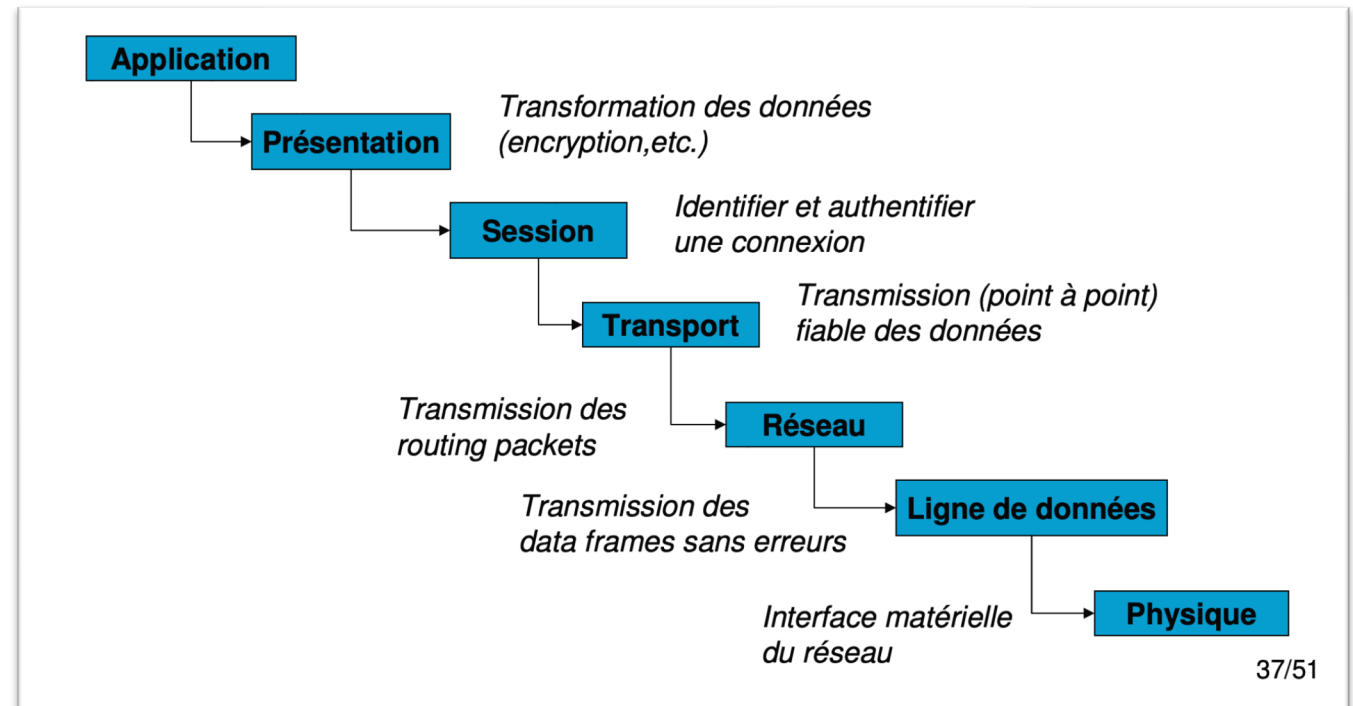
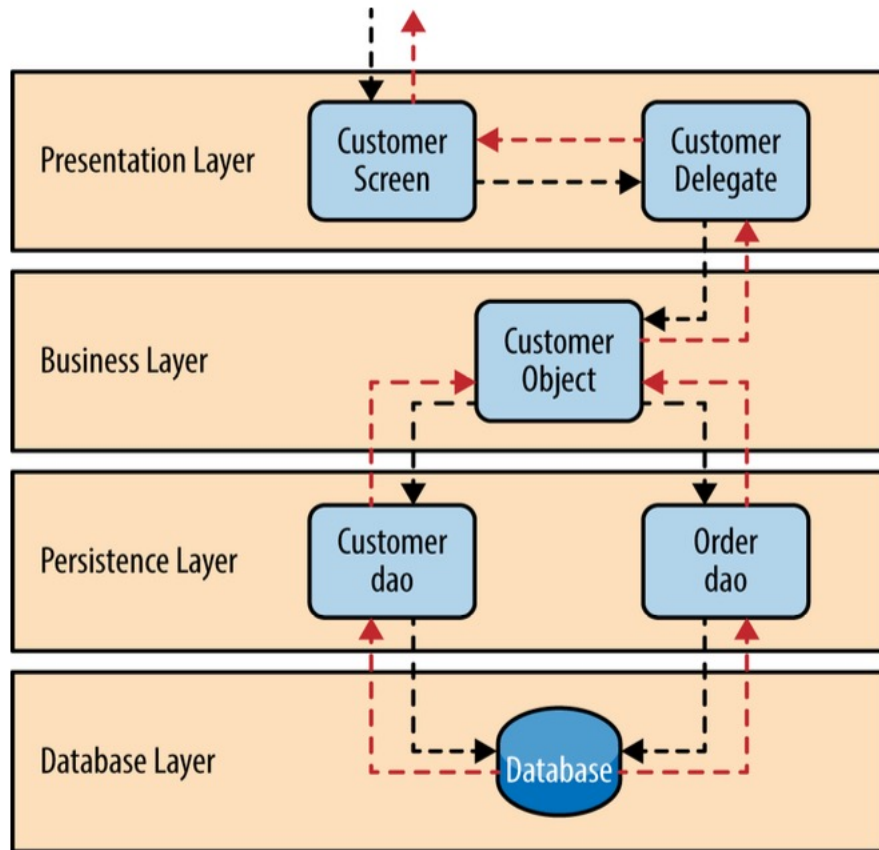


# Exemples typique de couches

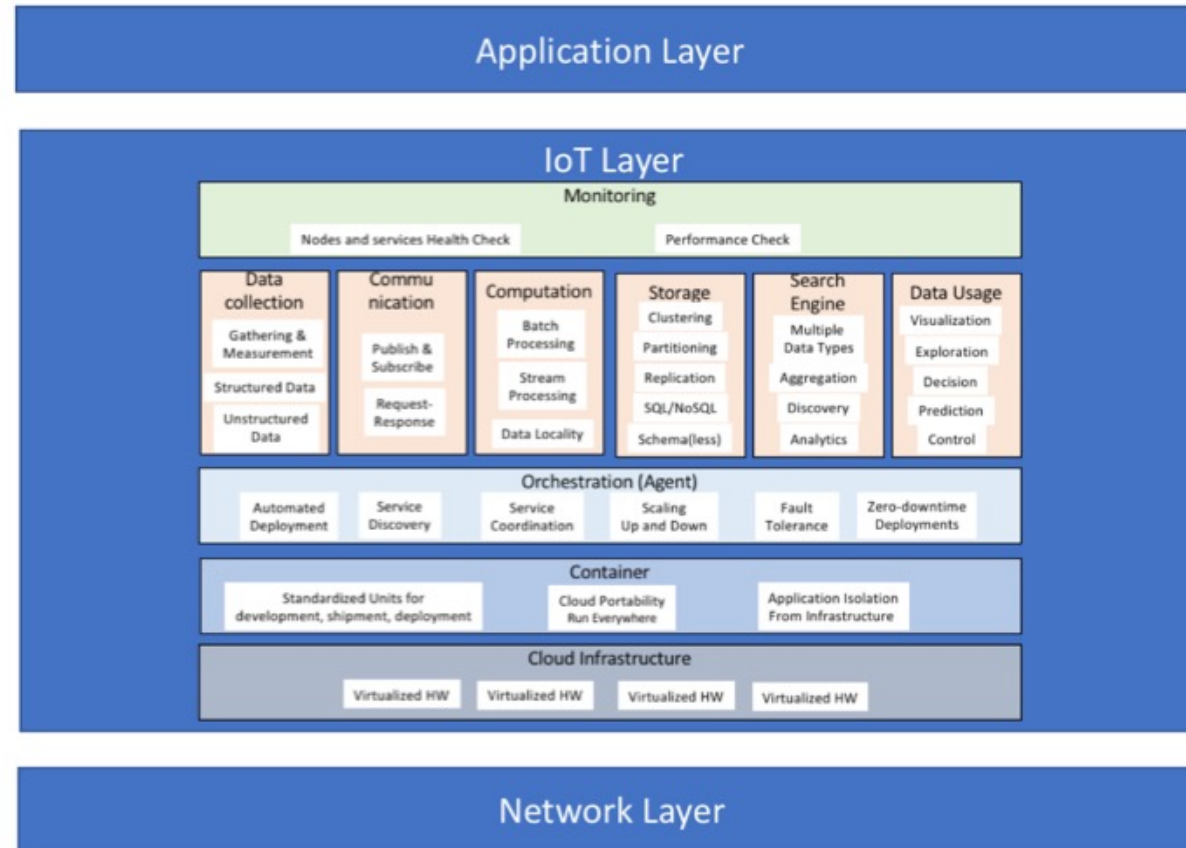
---

- 1. Couche Presentation (interface) :** Eléments d'interface pour interagir avec l'utilisateur.
- 2. Couche Business (logique applicative) :** Permet de faire les traitements, vérifications, notifications requises par l'application.
- 3. Couche Persistence :** Abstraction et Manipulation des données stockées dans la couche Database pour qu'elles soient facilement consommable dans la couche Business.
- 4. Couche Database (stockage) :** réalise le stockage et permet la recherche et la récupération de données.

# Exemples



# Exemples : Multi-couches et IoT





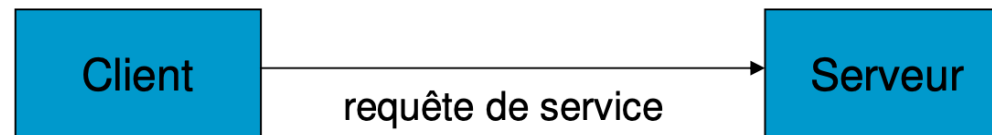
# Dans les systèmes distribués

---

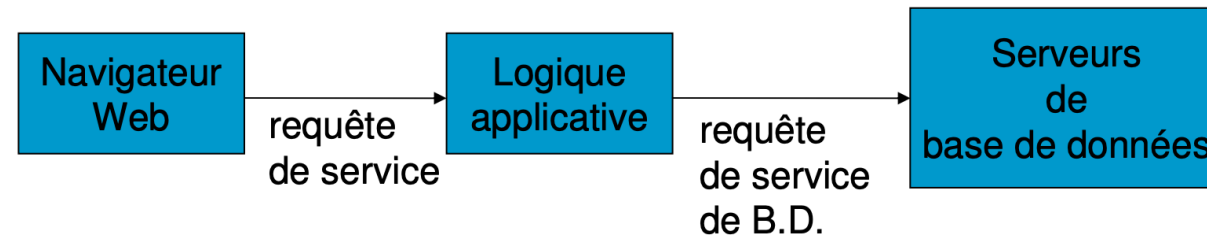
- Chaque couche agit comme un :
  - Serveur (pour les couches supérieures).
  - Client (consomme les couches inférieures).
  - Des connecteurs forment les protocoles entre les couches :
    - Et reposent sur des protocoles de communications (pas toujours du pull!)
- Les architectures client serveur simples sont aussi des architectures 2-niveaux.
- Les applications web LAMP sont souvent des architectures 3-niveaux.

# N-tiers

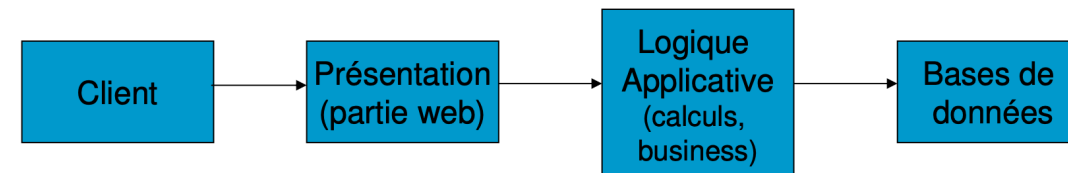
- Architecture 2-tiers



- Architecture 3-tiers



- Architecture 4-tiers



# Pro and Cons

---

## Avantages

- Conception et testabilité des couches de manière séparée, facilite l'évolutivité.
  - Facile d'ajouter / retirer /modifier des couches
- Cohésion, une couche peut servir des services cohésif aux autres couches.
- Séparation des préoccupations et abstraction :
  - Faible couplage.
  - Réutilisable.

## Inconvénients

- Performances, parfois il est inefficaces de passer par toutes les couches.
- Passage à l'échelle pas toujours facile car la granularité des couches peut être grosse.
- Grosse granularité rend parfois l'évolutivité compliquée.

# Pour le TP

*WebSockets:*  
Echange de messages.  
Communication bidirectionnelle.  
Mode connecté.

```
<script type = "text/javascript">
  function WebSocketTest() {

    if ("WebSocket" in window) {
      alert("WebSocket is supported by your Browser!");

      // Let us open a web socket
      var ws = new WebSocket("ws://localhost:9998/echo");

      ws.onopen = function() {

        // Web Socket is connected, send data using send()
        ws.send("Message to send");
        alert("Message is sent...");
      };

      ws.onmessage = function (evt) {
        var received_msg = evt.data;
        alert("Message is received...");
      };

      ws.onclose = function() {

        // websocket is closed.
        alert("Connection is closed...");
      };
    } else {

      // The browser doesn't support WebSocket
      alert("WebSocket NOT supported by your Browser!");
    }
  }
</script>
```

Ouverture du websocket

Envoyer message

Event handler



# Event-Driven Architectures

---

- Un patron d'architecture qui permet de concevoir des applications qui **passent à l'échelle**.
  - Mais utilisable pour des applications **petites ou à large échelle**.
- Se compose de composants **découplés** les uns des autres qui reçoivent et traitent de manière **asynchrone** des **événements**.
- Principaux concepts :
  - **Event processor** : reçoit un événement et le traite. Découplé et indépendant, se concentre sur une préoccupation particulière. Un event processor ne devrait pas dépendre d'un autre.
  - **Événement** : représente un changement d'état. Comprend deux parties, une en-tête (e.g., type, topic, timestamp) et un message.

# Communications asynchrones

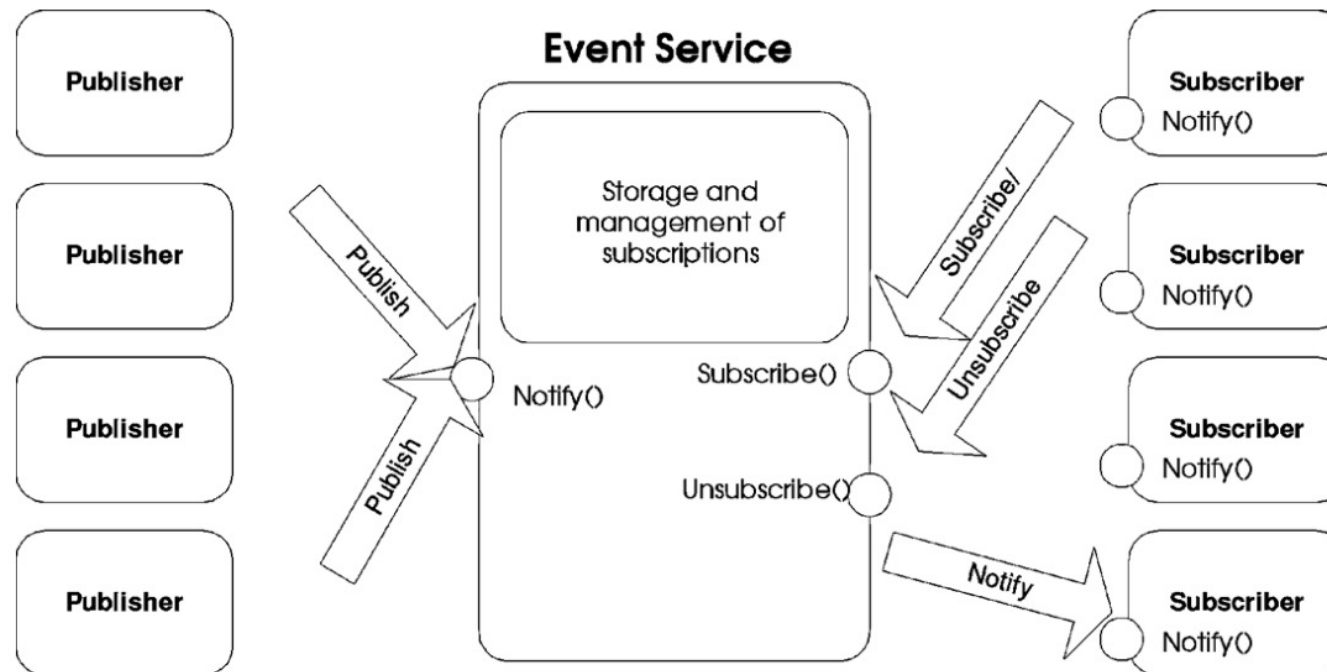
---

- **On n'est plus sur une approche requête - réponse** dans laquelle :
  - Le client « vient chercher » les requêtes à intervalle régulier (active polling).
  - Si on a N capteurs, le client doit envoyer N requêtes périodiquement afin de savoir si une valeur / mesure a changé depuis la dernière requête.
- Au contraire un événement est émis par un composant :
  - En réponse à un appel direct d'un client ou,
  - Comme le résultat d'un changement observé par le service.
  - Concrètement prend la forme d'un message ou d'une invocation de méthode.
- Le composant qui émet l'événement est appelé un **publisher**, et le client qui reçoit l'événement est appelé un **subscriber**.

# Publish / subscribe

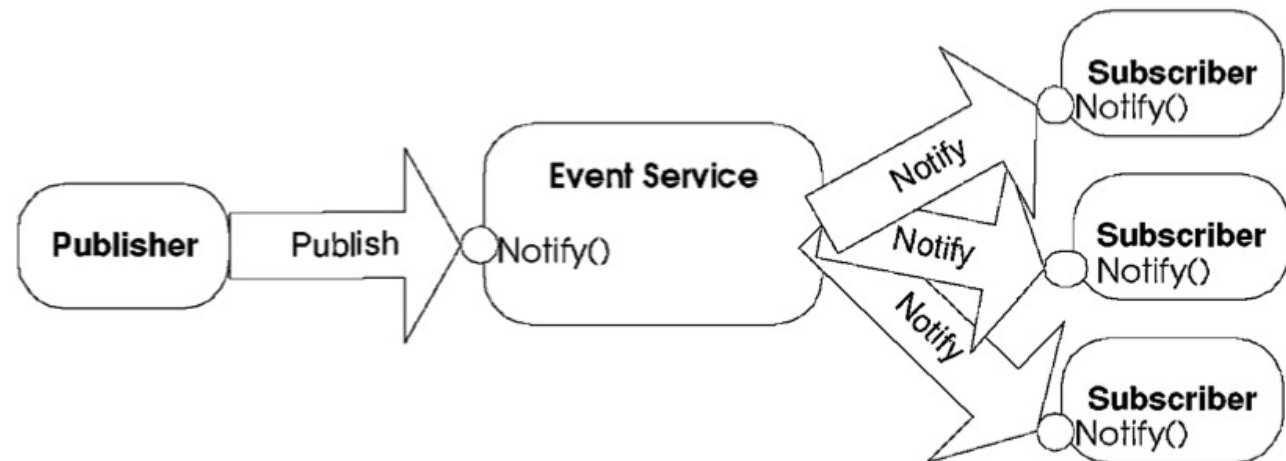
« Subscribers have the ability to express their interest in an event and are subsequently notified of any event, generated by a publisher, which matches their registered interest. An event is asynchronously propagated to all subscribers that registered interest in that given event. »

-- <https://infoscience.epfl.ch/record/165428/files/10.1.1.10.1076.pdf>



# Space decoupling

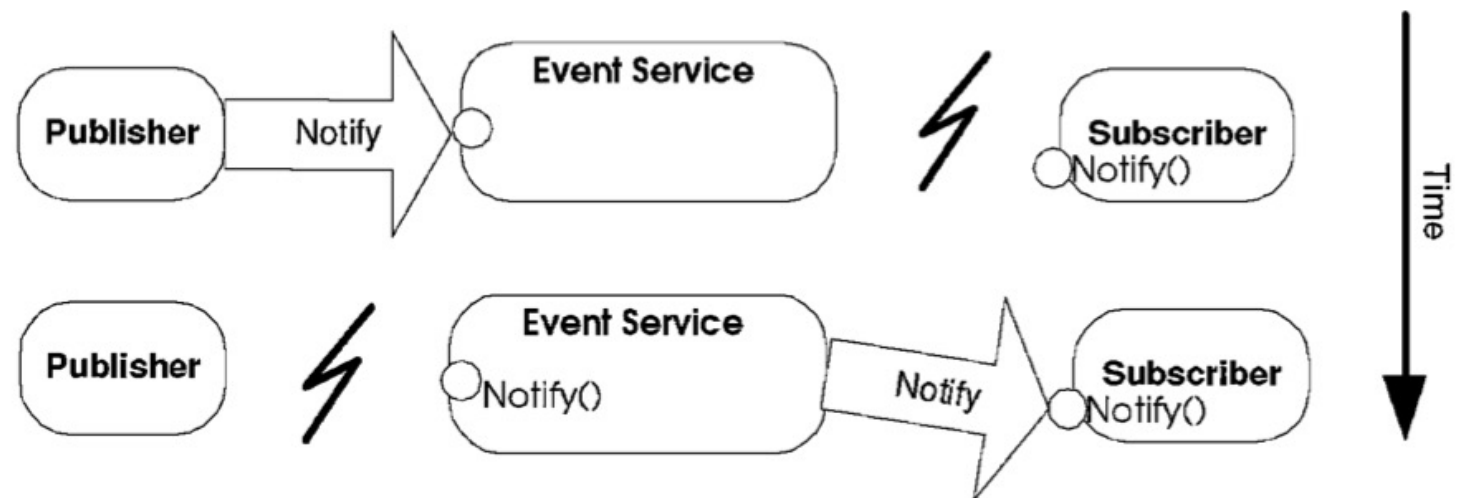
- **Les différents composants n'ont pas besoins de se connaître les uns les autres.**
- Les **publishers** publient des événements via un **event service** et les **subscribers** reçoivent ces événements indirectement via **l'évent service**.
  - Les *publishers* ne gardent pas de références vers les *subscribers* (et n'en connaissent pas le nombre).
  - Les *subscribers* ne gardent pas de références vers les *publishers* (et n'en connaissent pas le nombre).





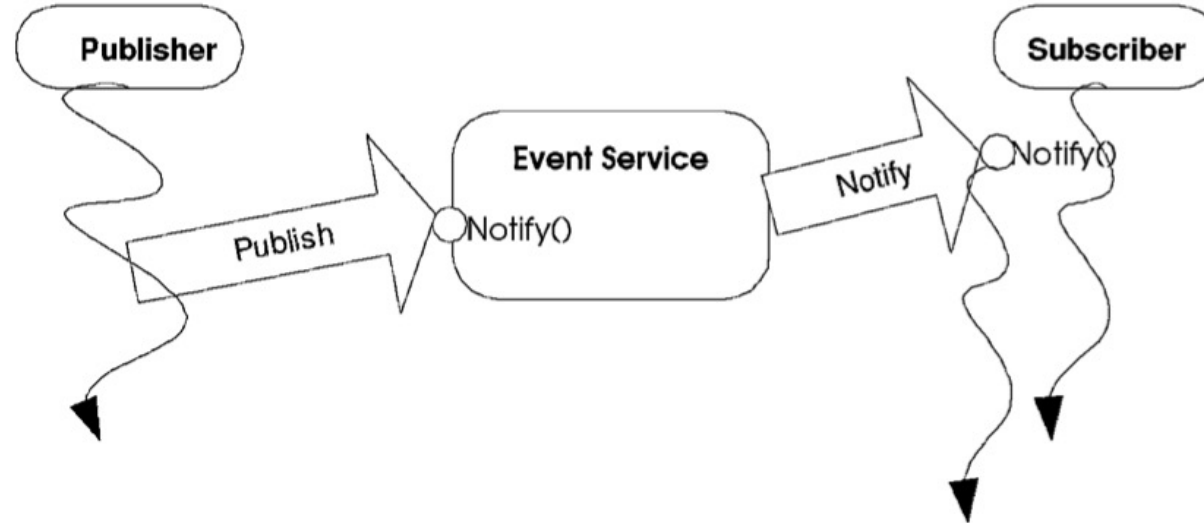
# Time decoupling

- **Les composants n'ont pas besoin d'être présent dans le système au même moment.**
  - Les *publishers* peuvent émettre des événements alors que des *subscribers* ne sont pas connectés.
  - Des *subscribers* peuvent être ajoutés alors que le *publisher* était déjà en service.
  - Les *subscribers* peuvent recevoir des événements alors que le *publisher* qui a émis l'événement n'est plus connecté.



# Synchronization decoupling

- **La production et la consommation d'événements étant asynchrone.**
  - Les *publishers* ne sont pas bloqués lors de l'émission d'événements.
  - Les *subscribers* peuvent être notifiés de manière asynchrone de l'occurrence d'un événement. Le traitement de l'événement peut être traité concurremment.



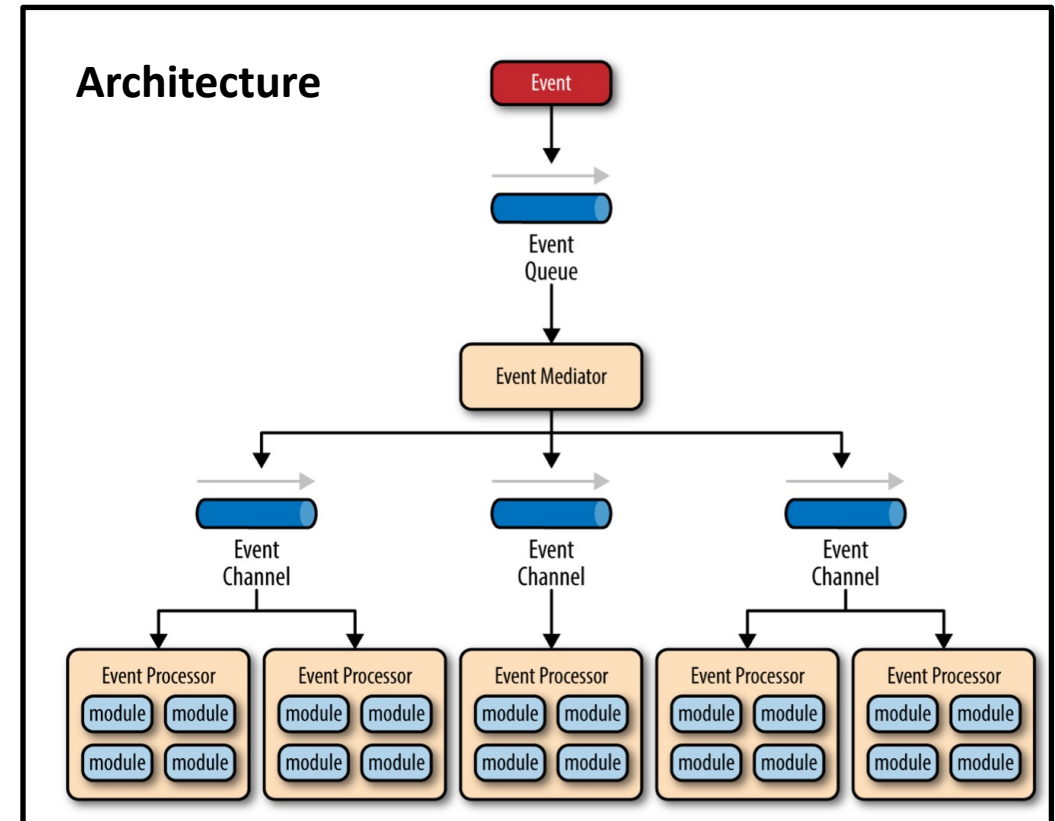
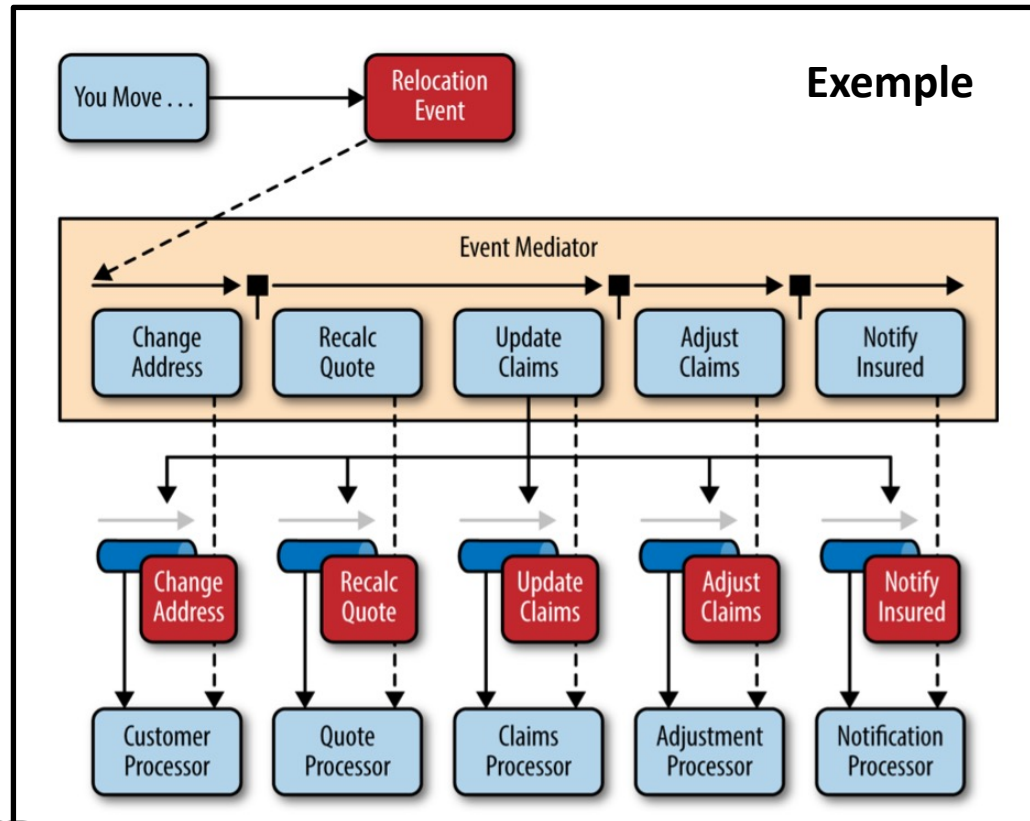
# Ce que cela implique pour l'évent processor

---

- **Intéropabilité** : du moment qu'il connaît le protocole de communication (y compris format/structure des données), **chaque event processor peut** :
  - être codé dans n'importe quel langage,
  - s'exécuter sur n'importe quelle plateforme (matérielle et logicielle).
- Un **event processor doit** :
  - s'abonner aux événements qu'il souhaite traiter.
  - traiter les événements auquel il est abonné sans préjuger d'un quelconque ordre et émettre un événement compte rendu de l'action qu'il vient de réaliser.
  - être autonome.

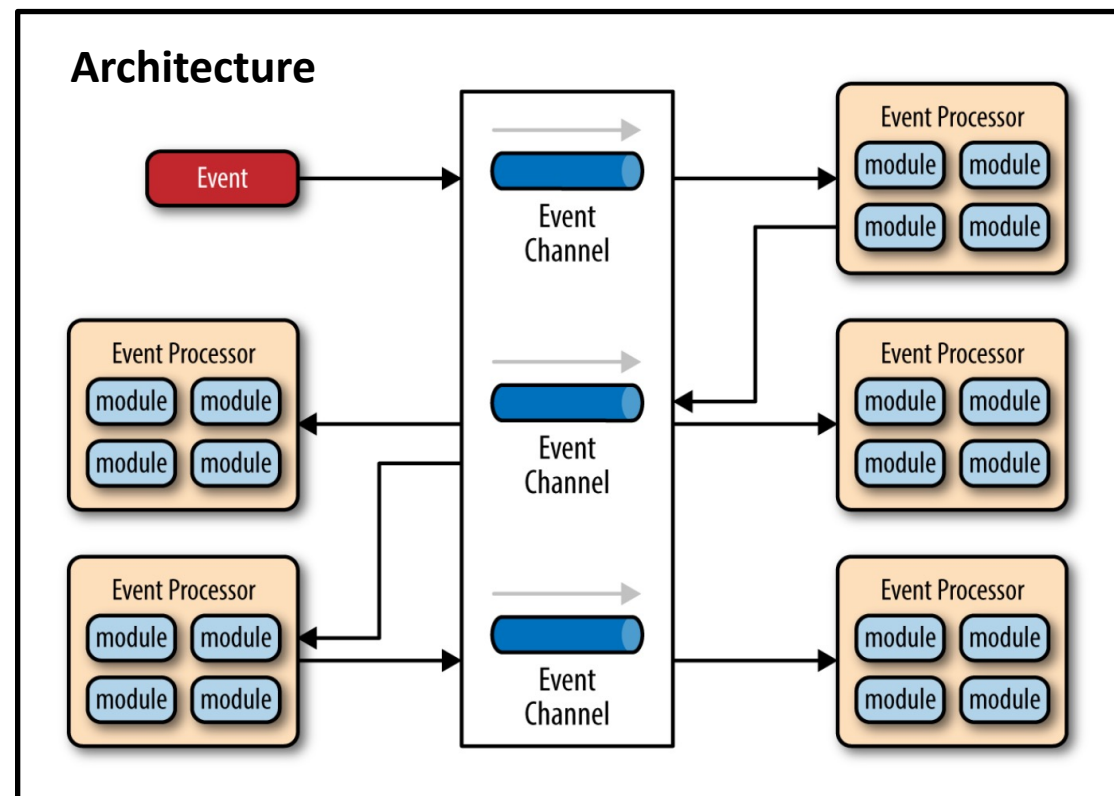
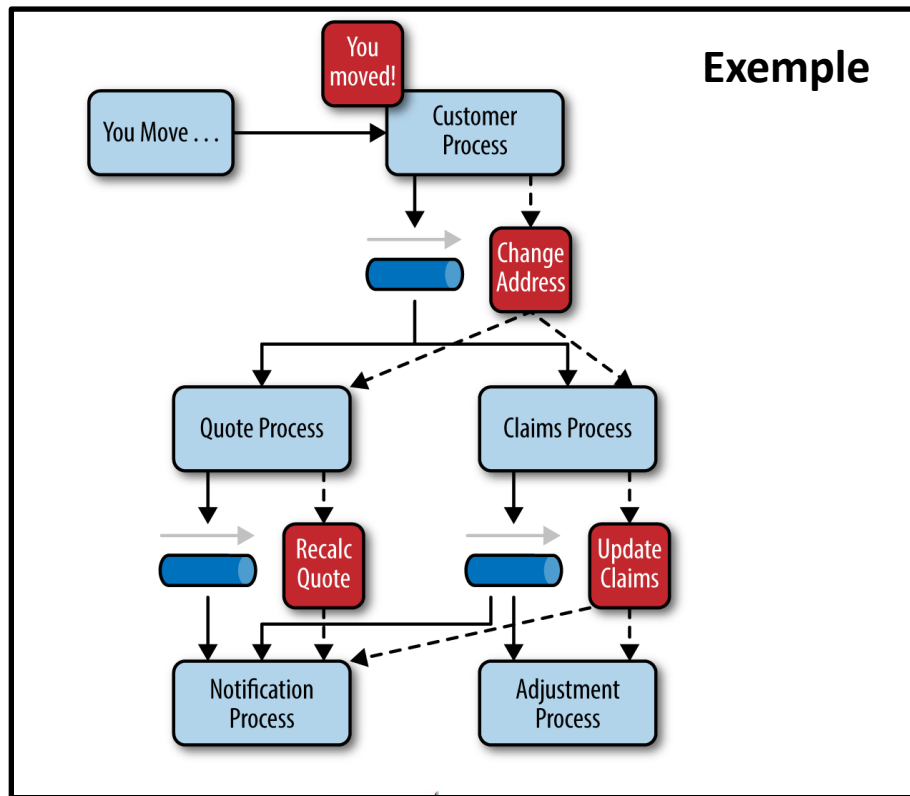
# Mediator topology

- The mediator topology is commonly used when you need to orchestrate multiple steps within an event through a central mediator.



# Broker topology

- The broker topology is used when you want to chain events together without the use of a central mediator.



# Pro and cons

---

## Avantages

- Facile à faire évoluer via découplage temporel et spatial.
- Réutilisabilité des event processors.
- Passage à l'échelle et performance (facile de faire passer à l'échelle des event processor).

## Inconvénients

- Complexité du développement car approche asynchrone.
- Tester un event processor est simple mais tester un ensemble peut être plus compliqué.

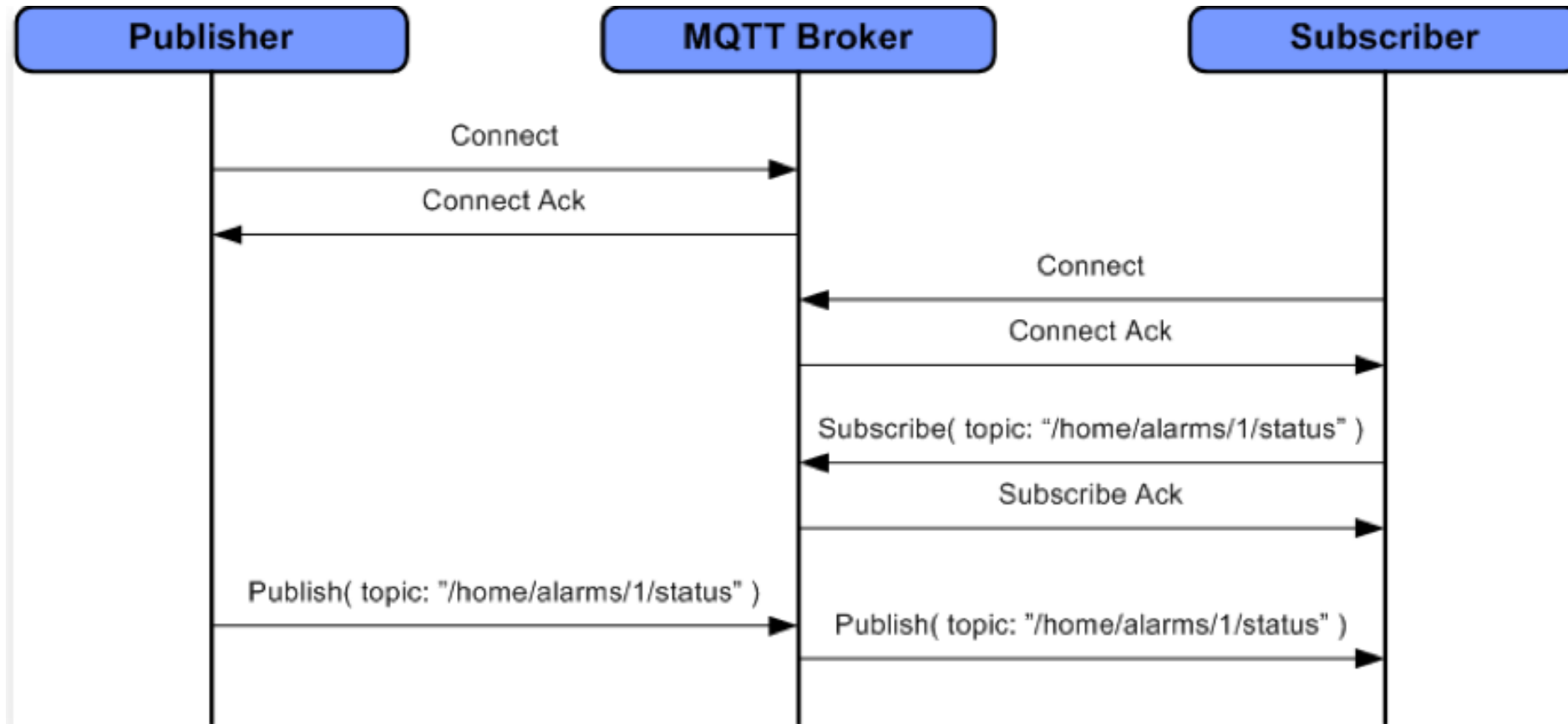
# MQTT (Message Queue Telemetry Transport)

---

- Un protocole publish/subscribe léger plus particulièrement dédié à l'IoT, M2M, etc.
  - Peut être utilisé par des dispositifs à faible capacités calculatoires.
    - MQTT control packet headers are kept as small as possible.
- Standard OASIS :
  - [https://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=mqtt](https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt)



# MQTT broker between publishers & subscribers





# Topic

---

- Les messages sont publiés sur des topics.
- Topics sous la forme d'un espace de nommage.
  - Hiérarchique avec chaque "sub topic" séparé par un /
    - `<country>/<region>/<town>/<postcode>/<house>/solarEnergy`
    - `<country>/<region>/<town>/<postcode>/<house>/alarmState`
- Un **subscriber** peut souscrire à un topic avec des wildcards :
  - Single-level wildcards "+" : n'importe où dans le topic.
  - Multi-level wildcards "#" : uniquement à la fin du topic.
  - **Only subscribers.**

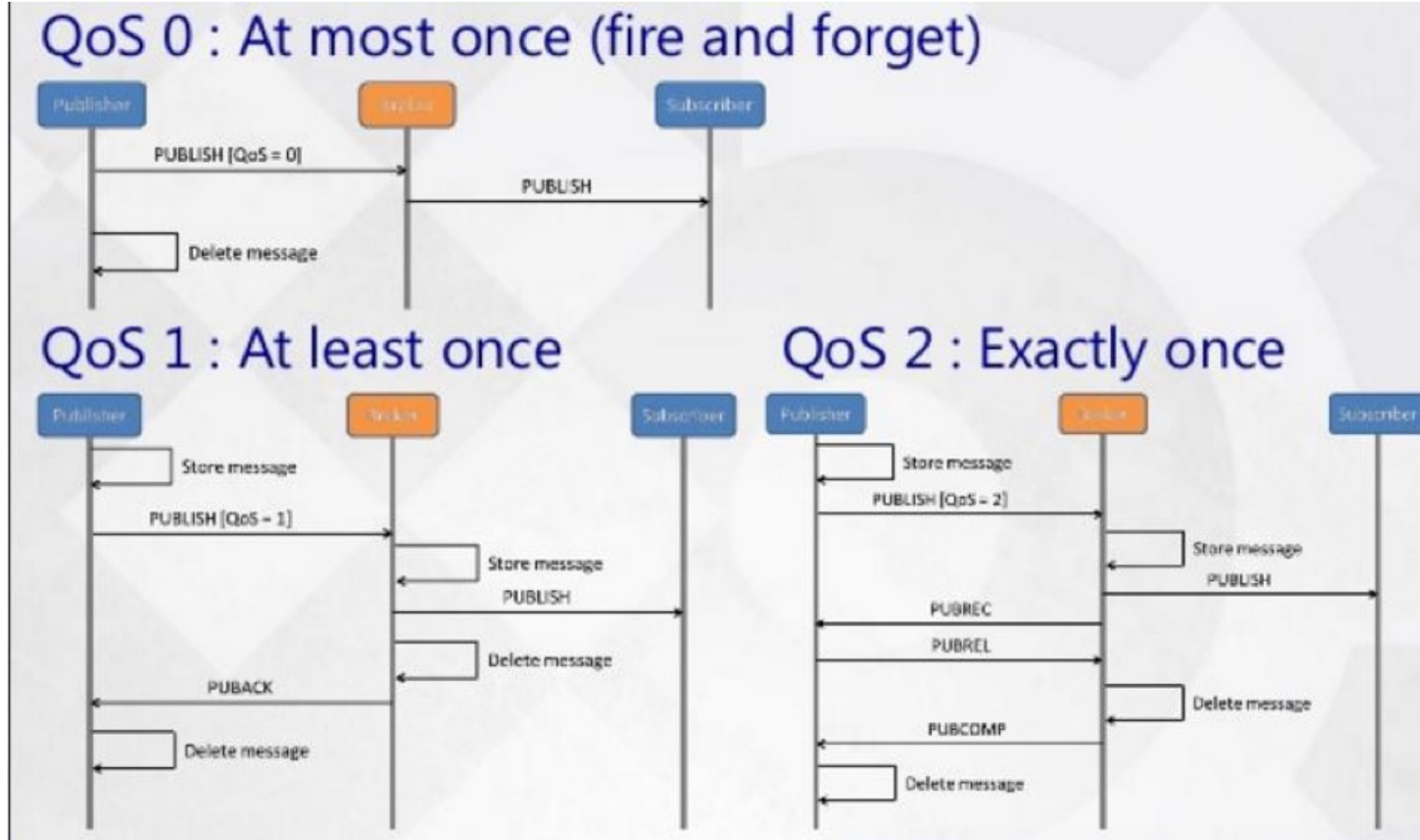
# QoS (Quality of Service)

---

- Quality of service (QoS) levels **determine how each MQTT message is delivered** and must be specified for every message sent through MQTT.
- **Three QoS** for message delivery could be achieved using MQTT:
  - QoS 0 (**At most once**) - where messages are delivered according to the best efforts of the operating environment. Message loss can occur.
  - QoS 1 (**At least once**) - where messages are assured to arrive but duplicates can occur.
  - QoS 2 (**Exactly once**) - where messages are assured to arrive exactly once.

***"The higher the QoS, the lower the performance".***

# QoS



# Une introduction aux SOAs

Juste une introduction, il y a beaucoup d'autres choses à dire !

# SOA

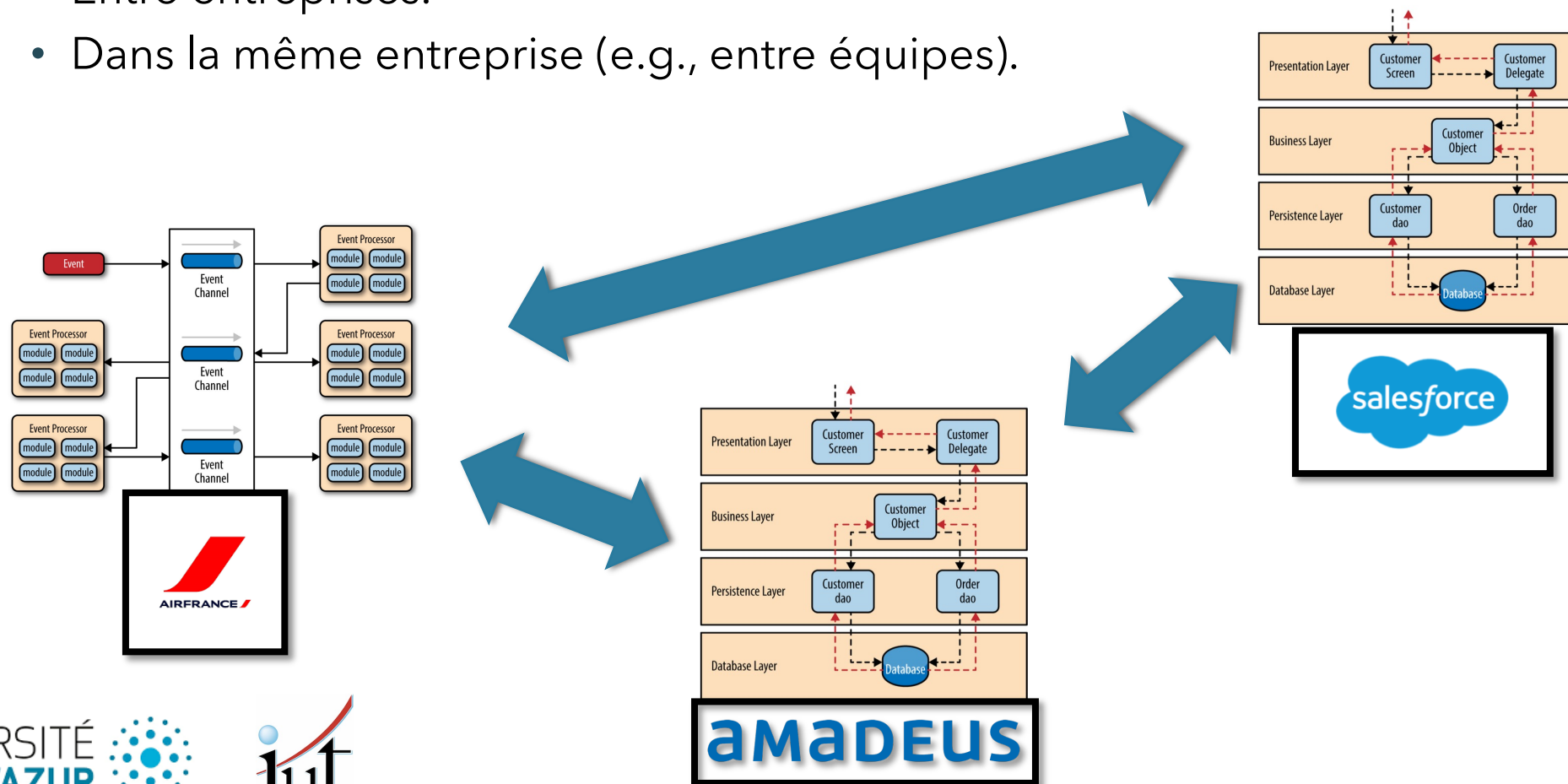
---

« *Service Oriented Architecture is a paradigm for organizing and utilizing **distributed** capabilities that may be under the control of **different ownership domains**. It provides a uniform means to **offer, discover, interact with** and use capabilities to produce desired effects consistent with **measurable preconditions** and **expectations**.»*

-- Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter Brown, and Rebekah Metz.  
*Reference model for service oriented architecture 1.0. Technical report, OASIS, 2006.*

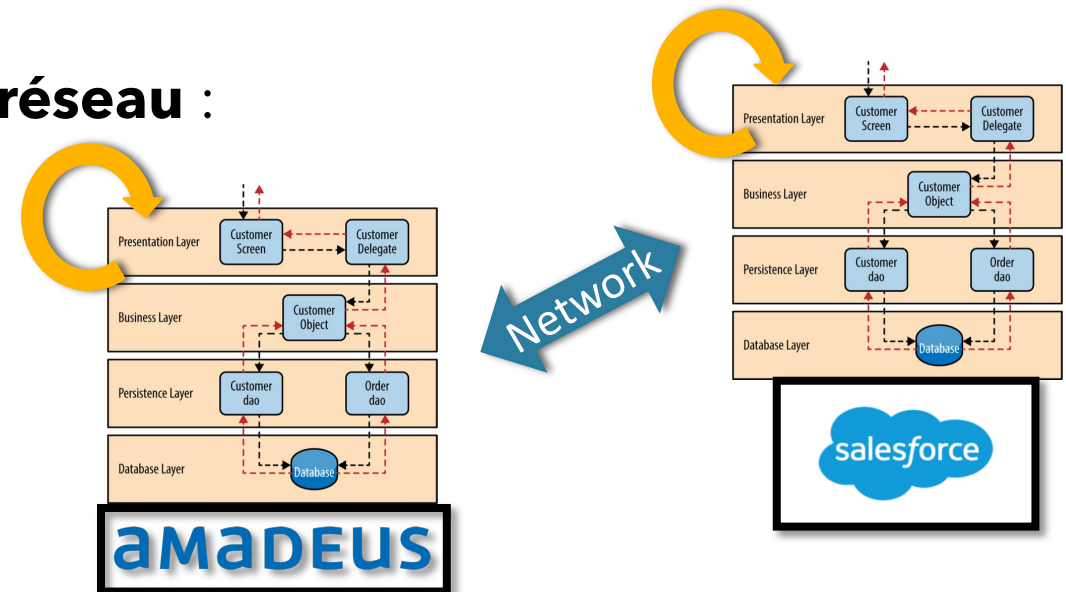
# Service Oriented Architecture (SOA)

- **Challenge de départ** : intégration, réutilisabilité et interopérabilité.
  - Entre entreprises.
  - Dans la même entreprise (e.g., entre équipes).



# Intégration

- Applications / systèmes développés par différentes entités :
    - Qui **évoluent à des rythmes différents** et de manière non coordonnée.
    - Qui reposent sur des **technologies différentes**
    - Qui répondent à des **problématiques différentes**.
    - Qui ont des **besoins différents**.
- } Ne facilite pas la réutilisabilité entre applications.
- Systèmes distribués qui interagissent via **réseau** :
    - Pb de sécurité.
    - Problème de performances.



# SOA

---

- Pour répondre à cette problématique d'intégration.
  - Y compris via les préoccupations présentées dans le premier cours et en particulier :
    - **Cohésion**
    - **Faible couplage**
- Qui sont des facilitateurs pour réutilisabilité, composabilité, flexibilité, évolution, passage à l'échelle et ... l'intégration !*



# Qui dit SOA dit **Services** !

---

- **Attention : Service  $\neq$  Web Service**

- Les web services sont une manière d'implémenter des services en utilisant des technologies du Web.

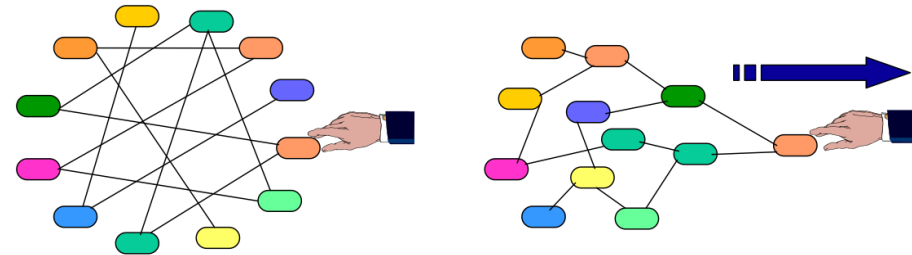
Plus facile pour s'exposer, se décrire aux consommateurs

- Entités logicielles orientées **business** - *i.e., une vision business du système.*
  - (et non pas technique comme peuvent l'être les composants.)
  - On cible les utilisateurs / clients (au sens customer) / consommateurs du service.
  - Un service regroupe un ensemble de fonctionnalités qui ont du sens (**cohésion**).
  - **Faiblement couplés** les uns aux autres.



# Composants & Services

- **Les deux ne sont pas incompatibles !**
- Ce que les services partagent avec les composants :
  - Boîtes noires.
  - Granularité.
  - Composables.
- Mais pas le même objectif.
  - Service = vision business;
  - Objet & Composants = vision technique.
- Les aspects dans les transparents suivants ne sont pas toujours implémentés avec les composants.



*Couplage fort = faible réutilisation,  
difficile à faire évoluer donc intégration difficile*

# Caractéristique 1 : Encapsulation

---

- Vus par leurs consommateurs comme des **boîtes noires**.
  - **Ce qui est exposé par l'interface est ce que l'on veut offrir au client et pas nécessairement ce que l'on fait dans le services.**
  - *i.e., un consommateur sait le service qui lui sera rendu mais pas comment cela est réalisé.*
- Pour cela : exposent une interface publique parfois appelée **contrat**. Afin de conserver leur propriété de **boîte noire**, cette interface ne contient aucune information relative à leur implémentation.
- Un fournisseur ne peut pas présumer de la façon dont sera utilisé son service.

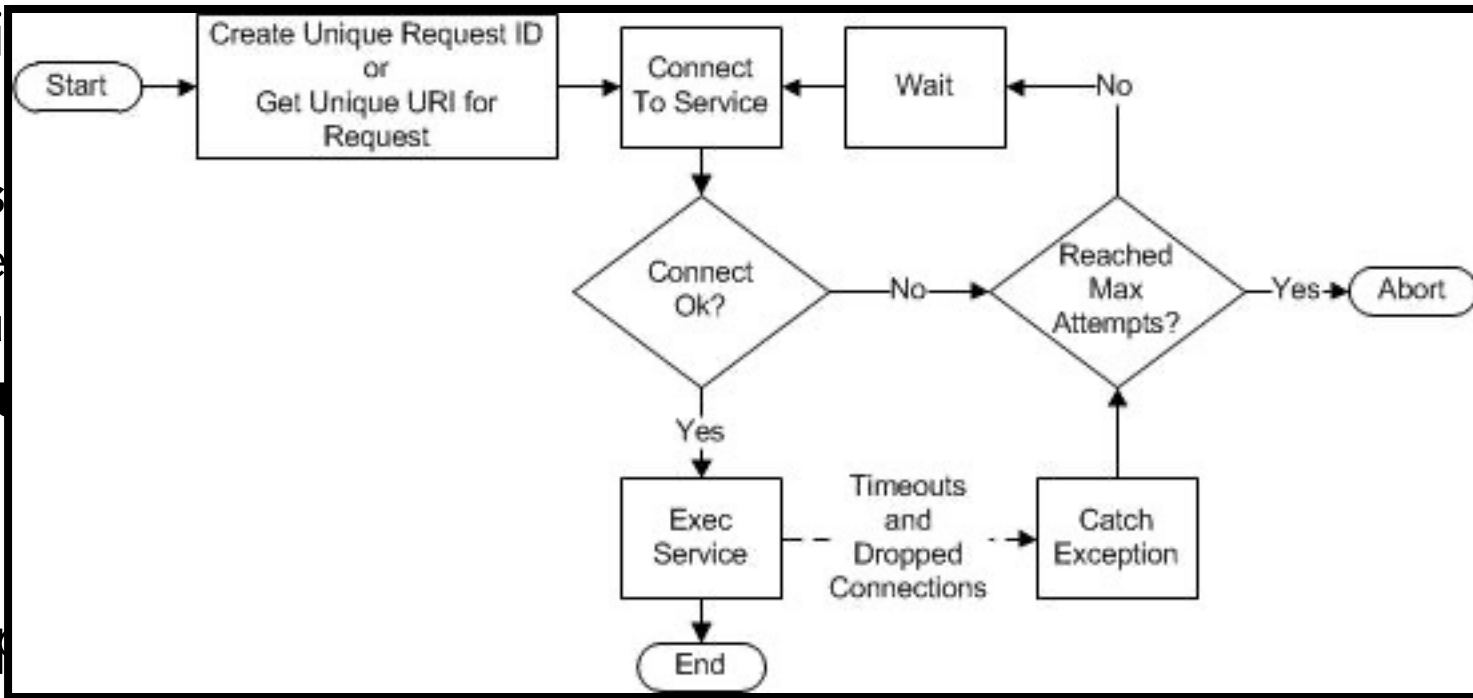
# Caractéristique 2 : Contrat

---

- Expose un **contrat** qui définit :
  - **Where and How** - *i.e., Comment utiliser le service ?*
    - Protocol de transport.
    - Protocol de communication.
    - Endpoint !
  - **Why** - *i.e., Quelles requêtes peuvent être réalisées ?*
    - Méthodes.
    - Ressources.
  - **What** - *i.e., Quelles sont les données requises et fournies ?*
    - Quel format.
    - Quel structure.
- Un service **doit se conformer à son contrat** puisqu'il s'agit là du seul moyen pour les autres entités d'obtenir des informations sur ses fonctionnalités.

# Caractéristique 3 : Autonomie

- Un service autonome est indépendant du reste de l'application.
  - **By design** : le service est conçu pour être indépendant (découplé).
  - **At runtime** : le service fonctionne de manière autonome.
- Dans le cas d'un service autonome (**stateless**).



ely from

ES

des erreurs.

er qu'un

e,

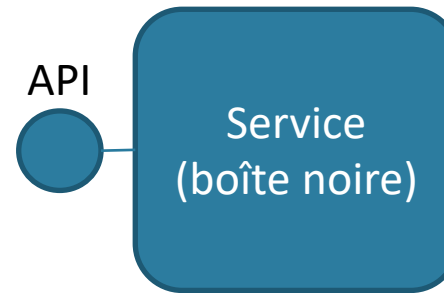
# Caractéristique 4 : Isolation

---

- Deux aspects distincts:
  - Un même service peut être découvert et utilisé par plusieurs consommateurs.
    - On veut **isoler** les consommateurs. Les requêtes d'un consommateur sont traitées indépendamment de celles d'un autre consommateur.
      - Préférable de faire des services **stateless**.
      - Préférable d'éviter les transactions.
  - Isoler les services entre eux.
    - L'échec de l'un n'entraîne pas la l'échec de l'autre.
    - Parfois jusqu'à avoir des environnements d'exécution isolés.

# API Styles

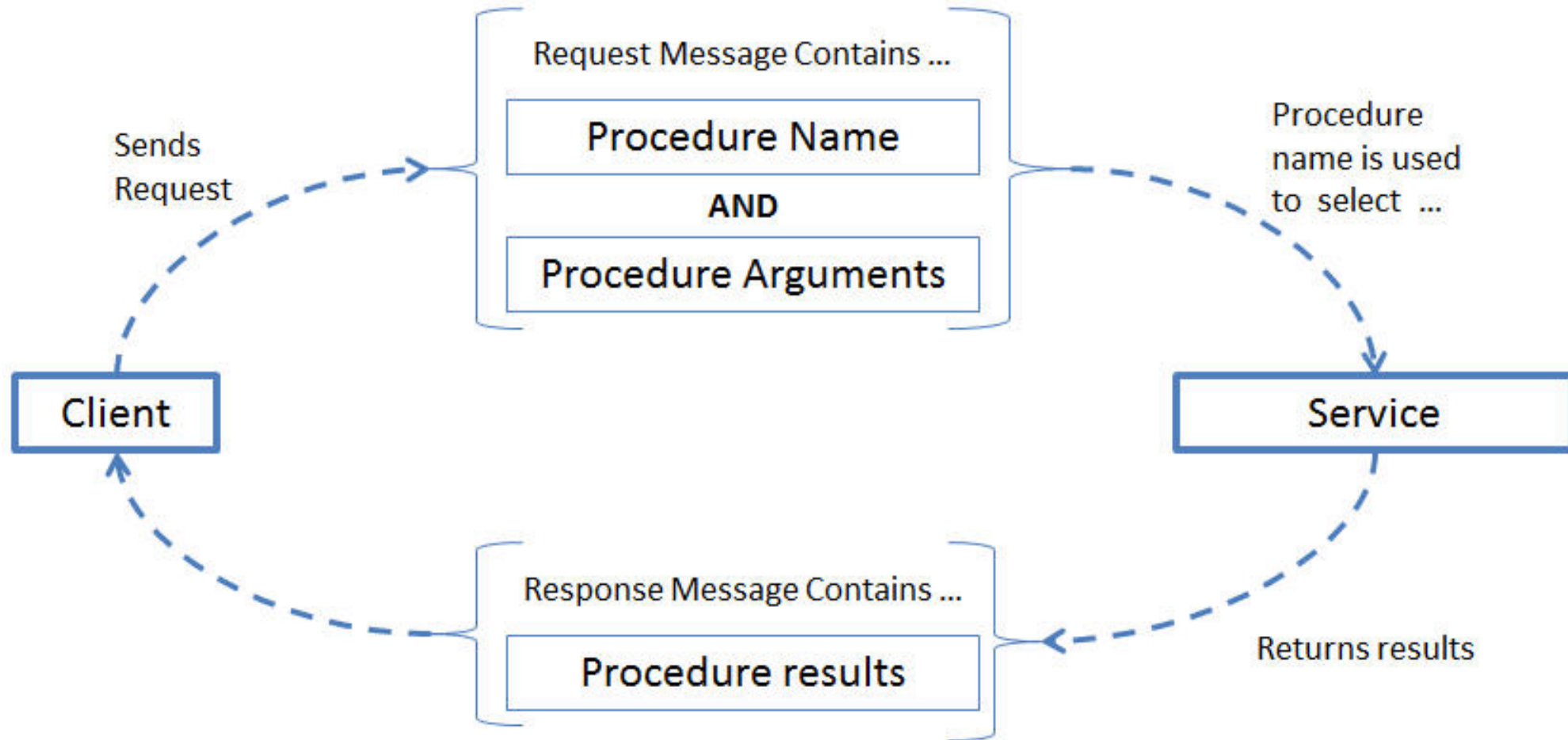
---



<u><a href="#">RPC API</a></u>	How can clients execute remote procedures?
<u><a href="#">Message API</a></u>	How can clients send commands, notifications, or other information to remote systems while avoiding direct coupling to remote procedures?
<u><a href="#">Resource API</a></u>	How can a client manipulate data managed by a remote system, avoid direct coupling to remote procedures, and minimize the need for domain-specific APIs?

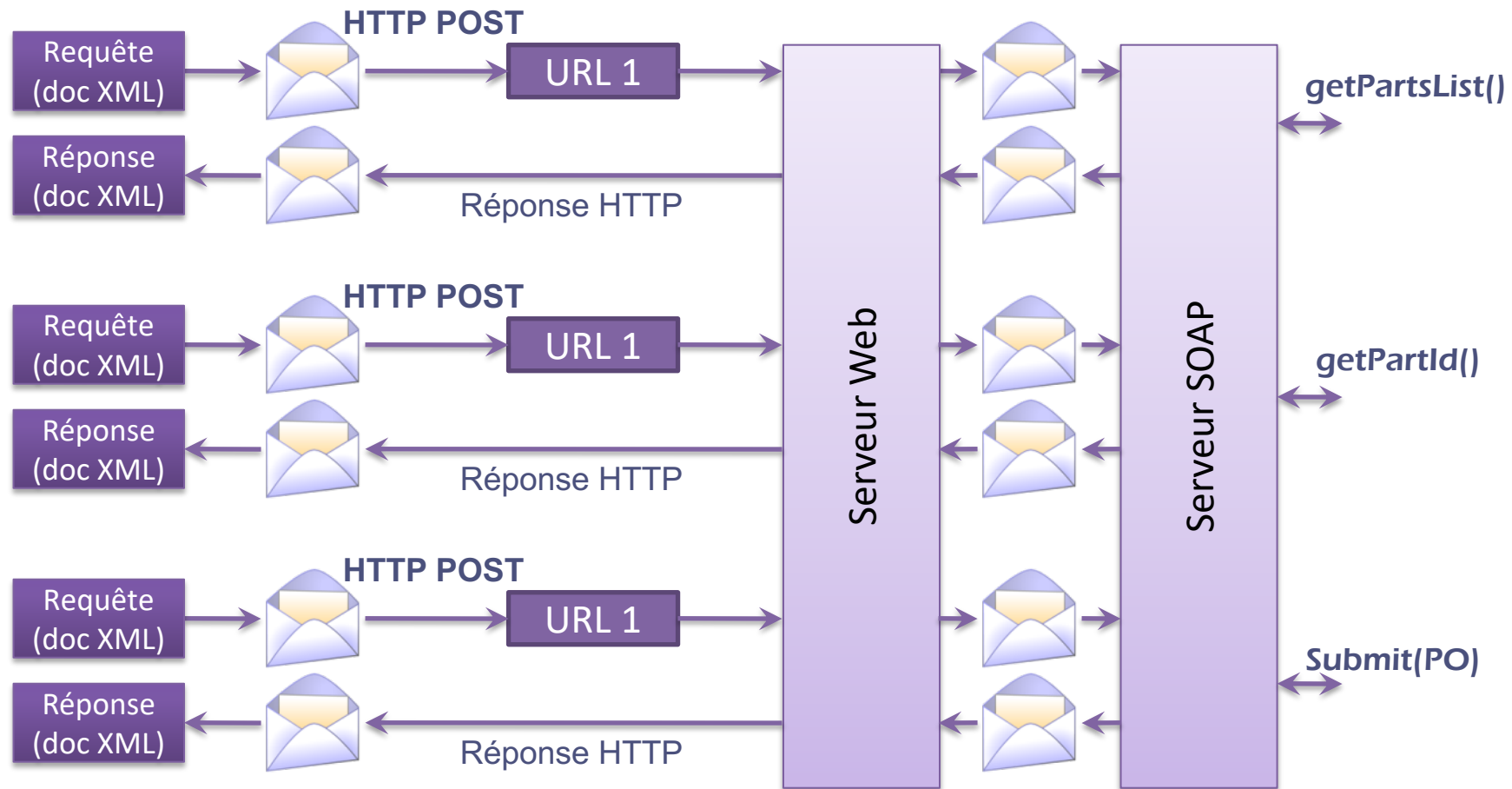
*Plus de détails dans le module programmation web services*

# Remote Procedure Call Style



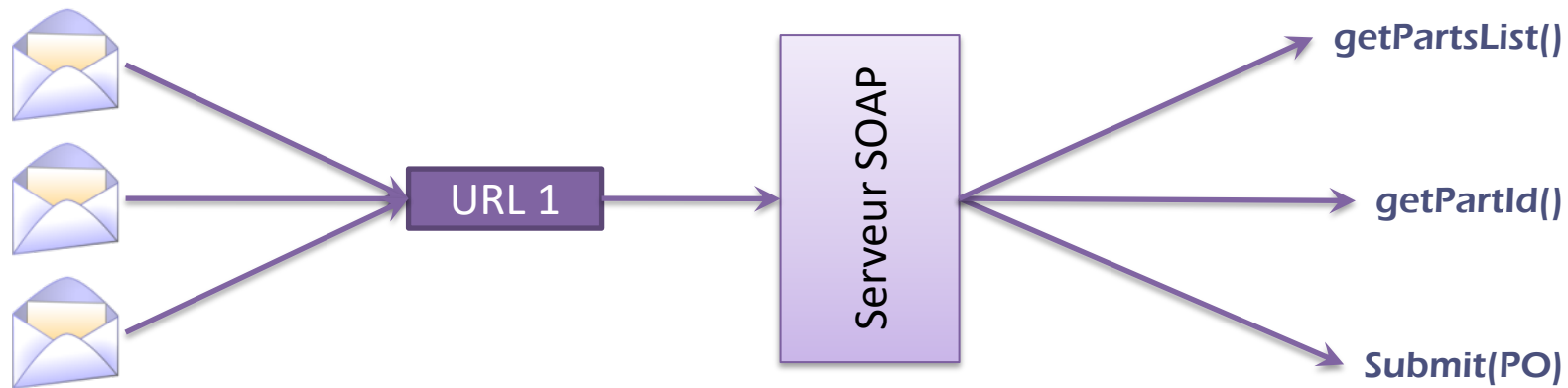


# Web Service - SOAP



# Tunneling

- Même si ce n'est pas une obligation, en SOAP, forme de tunneling sur la même URL



# SOAP Message

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAP-Action: "Some-URI"
```

Une seule commande  
HTTP/POST  
« envoi de message SOAP »

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

S  
O  
A  
P

# Contrat Web Service Description Language (WSDL)

← → ↻ soapapi.webservicespros.com/soapapi.asmx?WSDL ☆ 🌐 🗄 ⚙ 👤

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<?xml version="1.0" encoding="utf-8" />
<wsc:definitions xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:tns="http://tempuri.org/" xmlns:s1="http://microsoft.com/wsdl/types/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsd1="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://tempuri.org/">
  <wsc:types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
      <s:import namespace="http://microsoft.com/wsdl/types/" />
      <s:element name="GetProduct">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="ProductID" type="s:int" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetProductResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetProductResult" type="tns:Product" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="Product">
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="ProductID" type="s:int" />
          <s:element minOccurs="0" maxOccurs="1" name="Name" type="s:string" />
          <s:element minOccurs="0" maxOccurs="1" name="ProductNumber" type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="MakeFlag" nillable="true" type="s:boolean" />
          <s:element minOccurs="1" maxOccurs="1" name="FinishedGoodsFlag" nillable="true" type="s:boolean" />
          <s:element minOccurs="0" maxOccurs="1" name="Color" type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="SafetyStockLevel" nillable="true" type="s:short" />
          <s:element minOccurs="1" maxOccurs="1" name="ReorderPoint" nillable="true" type="s:short" />
          <s:element minOccurs="1" maxOccurs="1" name="StandardCost" nillable="true" type="s:decimal" />
          <s:element minOccurs="1" maxOccurs="1" name="ListPrice" nillable="true" type="s:decimal" />
          <s:element minOccurs="0" maxOccurs="1" name="Size" type="s:string" />
          <s:element minOccurs="0" maxOccurs="1" name="SizeUnitMeasureCode" type="s:string" />
          <s:element minOccurs="0" maxOccurs="1" name="WeightUnitMeasureCode" type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="Weight" nillable="true" type="s:decimal" />
          <s:element minOccurs="1" maxOccurs="1" name="DaysToManufacture" nillable="true" type="s:int" />
          <s:element minOccurs="0" maxOccurs="1" name="ProductLine" type="s:string" />
          <s:element minOccurs="0" maxOccurs="1" name="Class" type="s:string" />
          <s:element minOccurs="0" maxOccurs="1" name="Style" type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="ProductSubcategoryID" nillable="true" type="s:int" />
          <s:element minOccurs="1" maxOccurs="1" name="ProductModelID" nillable="true" type="s:int" />
          <s:element minOccurs="1" maxOccurs="1" name="SellStartDate" nillable="true" type="s:dateTime" />
          <s:element minOccurs="1" maxOccurs="1" name="SellEndDate" nillable="true" type="s:dateTime" />
          <s:element minOccurs="1" maxOccurs="1" name="DiscontinuedDate" nillable="true" type="s:dateTime" />
          <s:element minOccurs="1" maxOccurs="1" name="rowguid" nillable="true" type="s1:guid" />
        </s:sequence>
      </s:complexType>
    </s:schema>
  </wsc:types>
  <wsc:binding name="wsdl" type="tns:Product" />
  <wsc:port name="wsdl" binding="wsdl" />

```

# Pros & Cons

---

## Pros

- Très proches des approches de programmation classique.

## Cons

- Couplage aux procédures.
- Interface à plat.
- Tunneling.

# Exercice

---

- Shop owner:
  - **Manage Payment:**
    - All transactions from a customer.
    - All transactions of the day.
    - Process a payment.
  - **Manage customers:**
    - Manage existing customers.
    - Register new customers.
  - **Manage portfolio:**
    - Manage existing portfolio.
    - Delete existing products from the portfolio.
    - Add a product in the portfolio.

*Quels services définissez vous ?*

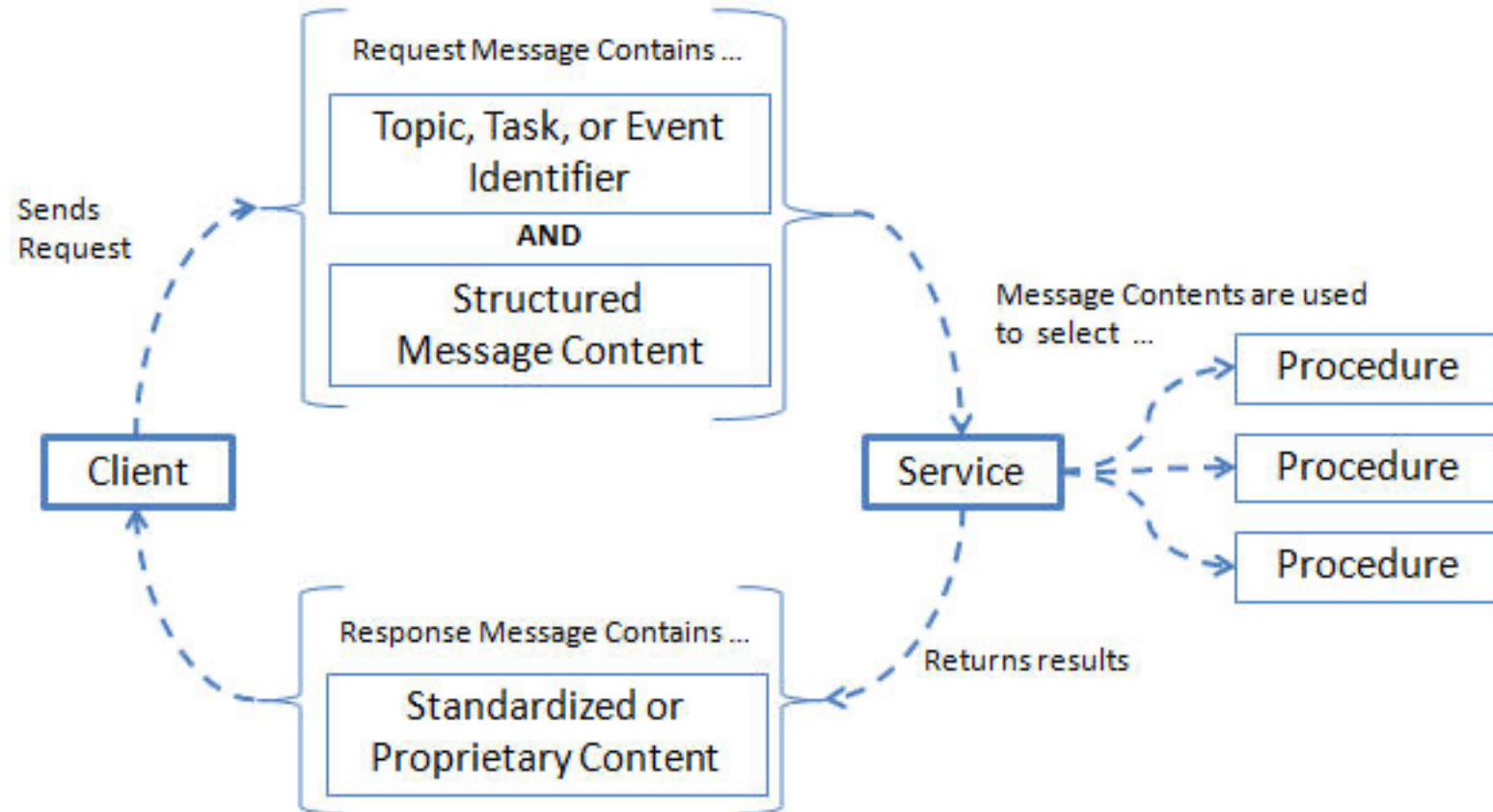
*Définissez les interfaces de ces services*

*RPC Style*

- 1. Procédures*
- 2. Arguments*
- 3. Retour*

- 
- Payment Service:
    - `get_all_transactions_per_customer(customerId)` : return all transactions from the customer
    - `Process_payment(sum, product, customerId)`: return the transaction status and id
    - `Get_all_transactions_per_day(start, end)`: return all dates within start and end.
  - Customer Service:
    - `Describe_customer(id)`:return a description of the customer
    - `Get_all_customers()`: return all ids of all customers
    - `Update_customer(id, infos...)`: return new customer description
    - `Add_customer(infos...)` : return customer id

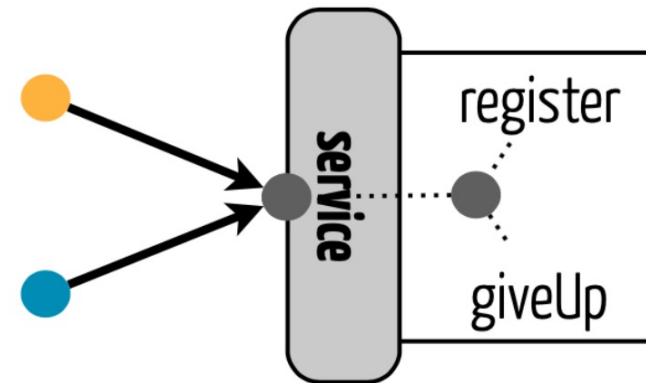
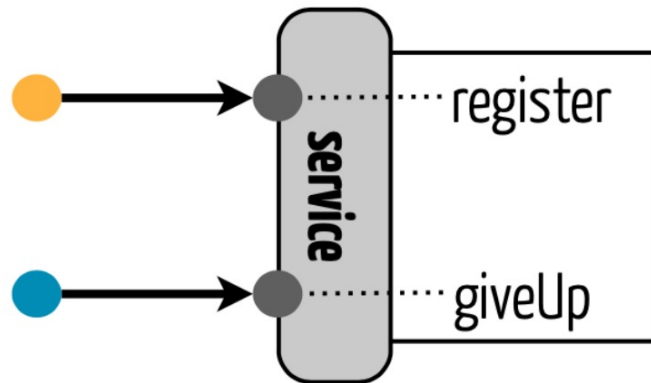
# Message Style





# Messages

- Meilleur découplage des procédures.
  - Le patron de conception délégation.



- L'action est choisie sur la base du contenu du message.
  - Un message peut contenir les informations pour plusieurs actions.
- Attention à ne pas multiplier les messages !

# Pros & Cons

---

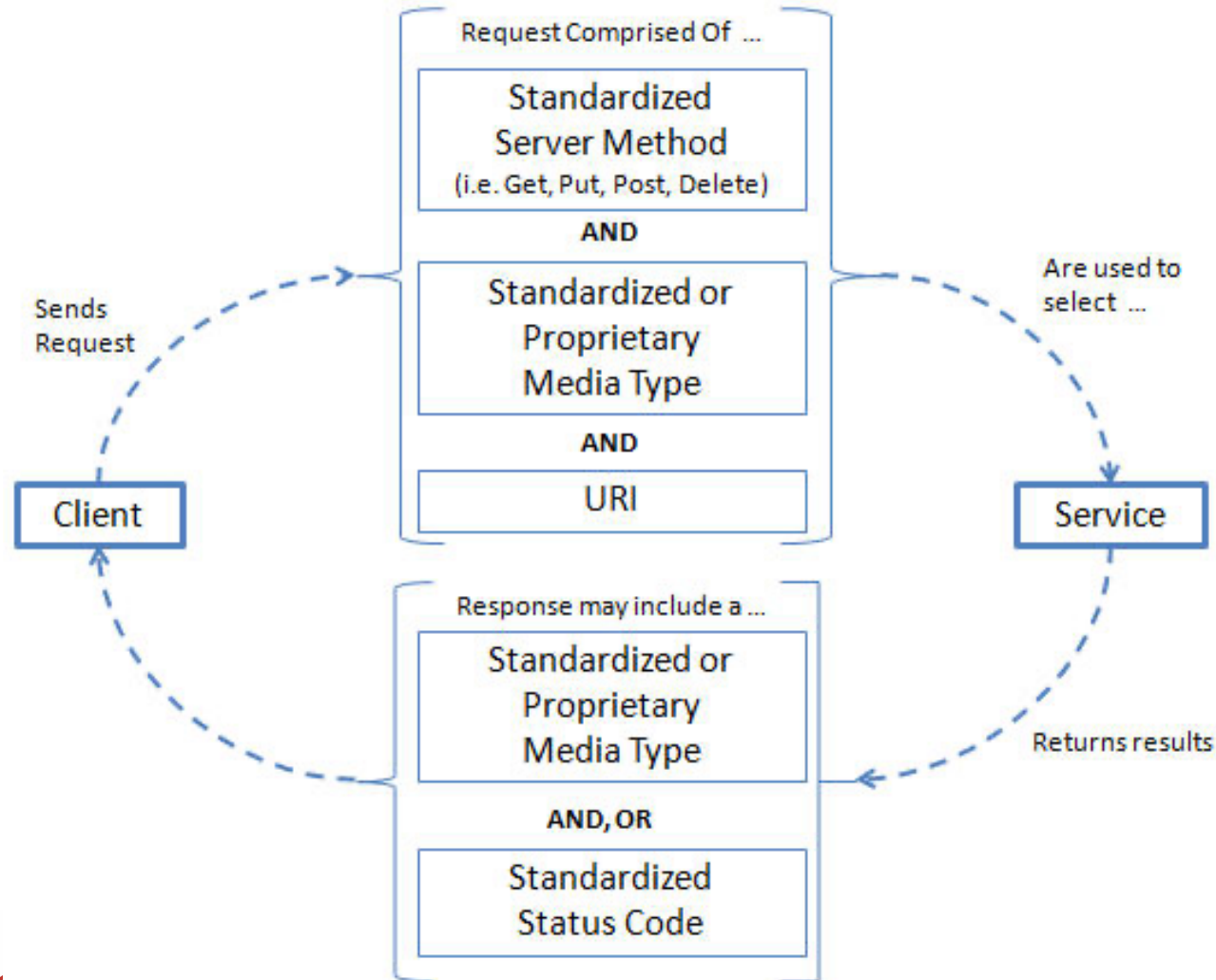
## Pros

- Meilleur découplage.
- Plus facile pour l'isolation.

## Cons

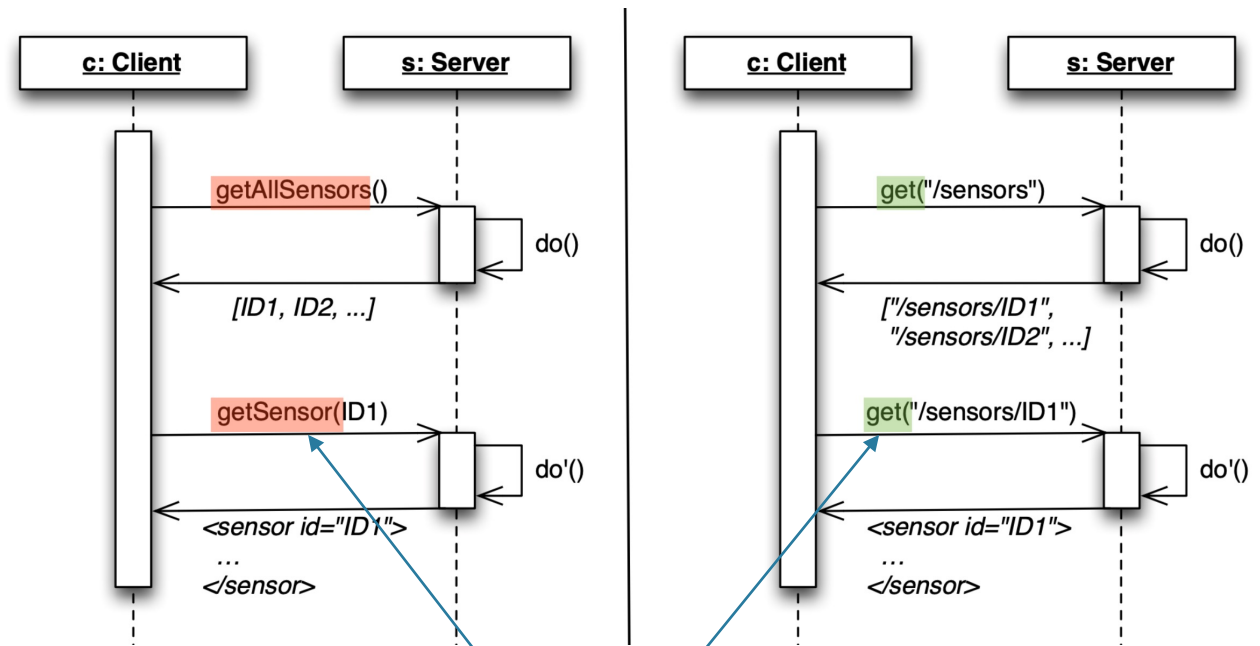
- Implémentation compliquée.
- On ne bénéficie pas de la description des méthodes.

# Resource Style



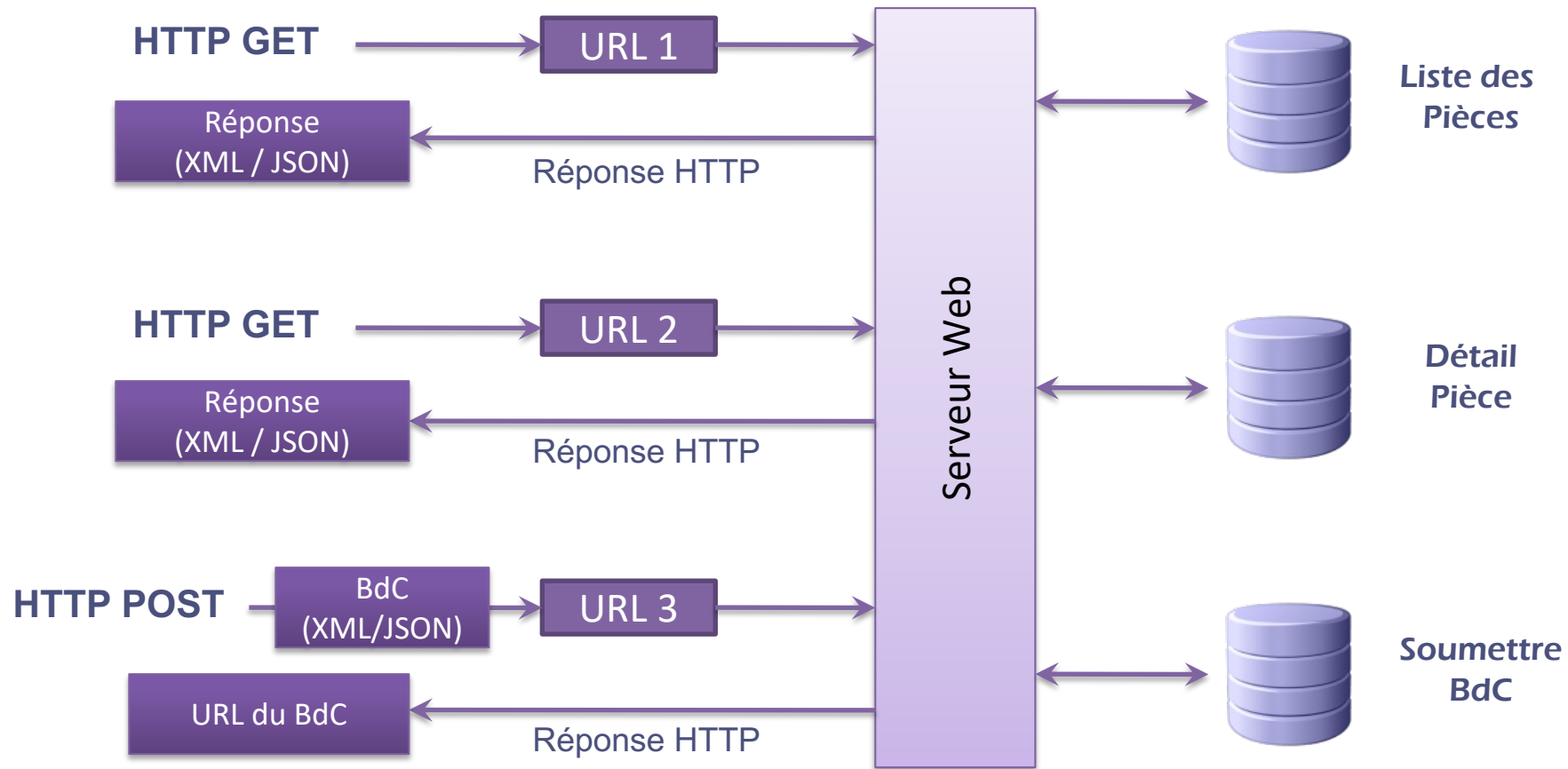
# Une autre perspective

- From **méthodes (actions)** to **resources (noms)**
  - Les clients accèdent à des ressources nommées de manière unique.



Différence  
Couplage

# Web Service REST



# Interface Uniforme

---

- GET `http://api.domain.com/users`
- POST `http://api.domain.com/users`
- PUT `http://api.domain.com/users/hugo`
- DELETE `http://api.domain.com/users/hugo`

Récupère les  
utilisateurs

Crée un nouvel  
utilisateur

Modifie hugo

Supprime hugo



# Exemple de message

---

```
POST http://MyService/Person/  
Host: MyService  
Content-Type: text/xml; charset=utf-8  
Content-Length: 123  
<?xml version="1.0" encoding="utf-8"?>
```

HTTP Header  
Commande POST

---

```
{  
  "ID": "1",  
  "Name": "M Vaqqas",  
  "Email": "m.vaqqas@gmail.com",  
  "Country": "India"  
}
```

HTTP Body  
JSON representation  
of a resource

# Contrat

## Open API - Swagger

The image shows the Swagger Editor interface. On the left, the OpenAPI specification is displayed in a code editor. On the right, the rendered API documentation is shown, including the title 'Swagger Petstore', the license 'MIT', the server URL 'http://petstore.swagger.io/v1', and a list of endpoints under the 'pets' resource.

```
1 openapi: "3.0.0"
2 info:
3   version: 1.0.0
4   title: Swagger Petstore
5   license:
6     name: MIT
7 servers:
8   - url: http://petstore.swagger.io/v1
9 paths:
10  /pets:
11    get:
12      summary: List all pets
13      operationId: listPets
14      tags:
15        - pets
16      parameters:
17        - name: limit
18          in: query
19          description: How many items to return at one
20            time (max 100)
21          required: false
22          schema:
23            type: integer
24            format: int32
25      responses:
26        '200':
27          description: A paged array of pets
28          headers:
29            x-next:
30              description: A link to the next page of
31                responses
32              schema:
33                type: string
34              content:
```

Swagger Petstore 1.0.0 OAS3

MIT

Servers

**pets** ▼

- GET** /pets List all pets
- POST** /pets Create a pet
- GET** /pets/{petId} Info for a specific pet

Models ▼



# Pros & Cons

---

## Pros

- Evite le tunneling.
- Interface uniforme.
- Pas de couplage aux procédures.
- Mise en cache de ressources.
- Facilite la mise en place isolation.

## Cons

- Le client doit conserver toutes les données nécessaires pour le bon déroulement d'un traitement.
  - Parfois complexe à mettre en œuvre.

# Exercice

---

- Shop owner:
  - **Manage Payment:**
    - All transactions from a customer.
    - All transactions of the day.
    - Process a payment.
  - **Manage customers:**
    - Manage existing customers.
    - Register new customers.
  - **Manage portfolio:**
    - Manage existing portfolio.
    - Delete existing products from the portfolio.
    - Add a product in the portfolio.

*Quels services définissez vous ?*

*Définissez les interfaces de ces services*

*Ressource Style*

*URI*

*VERB HTTP*

*Retour*

- 
- **/rest/payment/public/{id}/transactions:**
    - **GET:** returns the transactions associated to the customer identified by the given id
  - **/rest/payment/public/{id}/process:**
    - **POST:** the process used to perform a payment for customer id. If success status code 201.  
Object send with request: a JSON with sum, product
  - **/rest/customers:**
    - **GET:** returns a list of links to registered customers
    - **POST:** create a new customer (status code: 201), available as a new resource
  - **/rest/customers/{id}:**
    - **GET:** returns customer description
    - **PUT:** update a customer

# Et l'interopérabilité dans tout cela ?

- On a parlé d'entités distribuées reposant sur des plateformes, technologies et infrastructures hétérogènes ... mais comment peut on ajouter / enlever / faire communiquer des services entre eux et avec leurs consommateurs ?
- Réponse :
  1. Les **standards** !
    - Sur la description du contrat.
    - Sur le mécanisme pour découvrir des services.
    - Sur le protocole de communication.
  2. **Agreement** :
    - Sur ce qui est dans le contrat (why / what).

*Les Web Services proposent de construire des services sur la base des standards du web (HTTP, JSON, XML, etc.)*

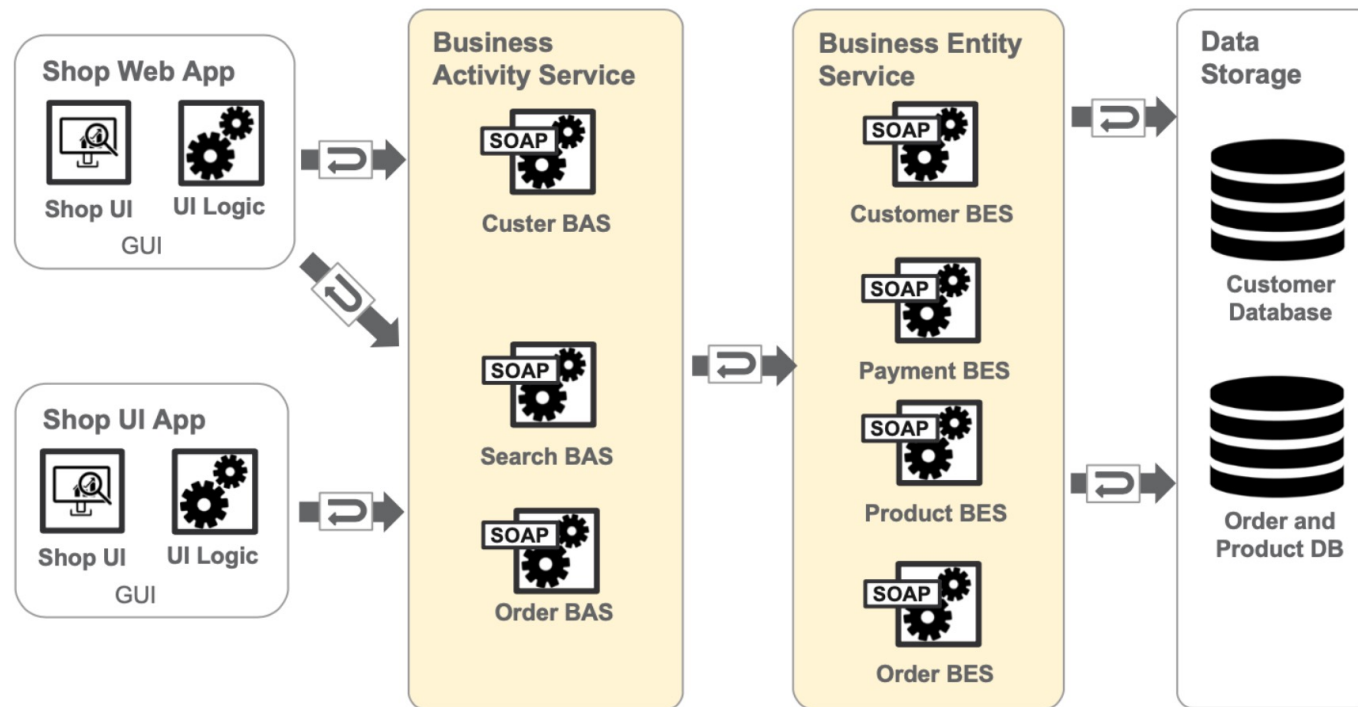
# Retour aux SOA

---

- Ok, on a des services mais comment construire une application à partir de ces services ?
- On va **composer** les services !

# Composition

- **Les services peuvent être coordonnés pour s'assembler et créer de nouvelles applications ou des services composites.**



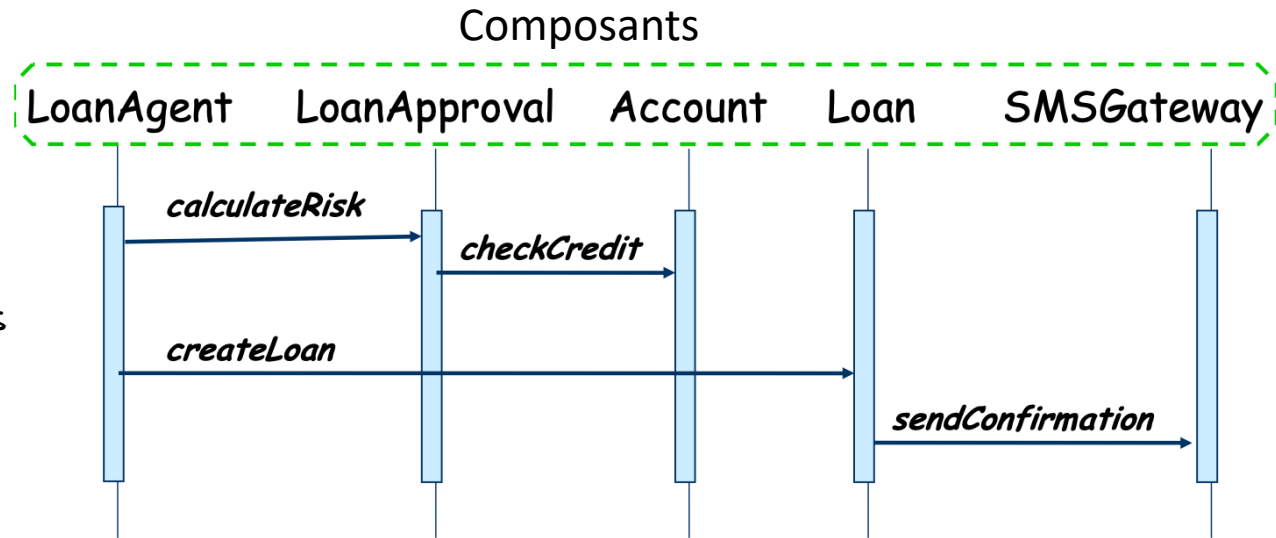
# Composition

---

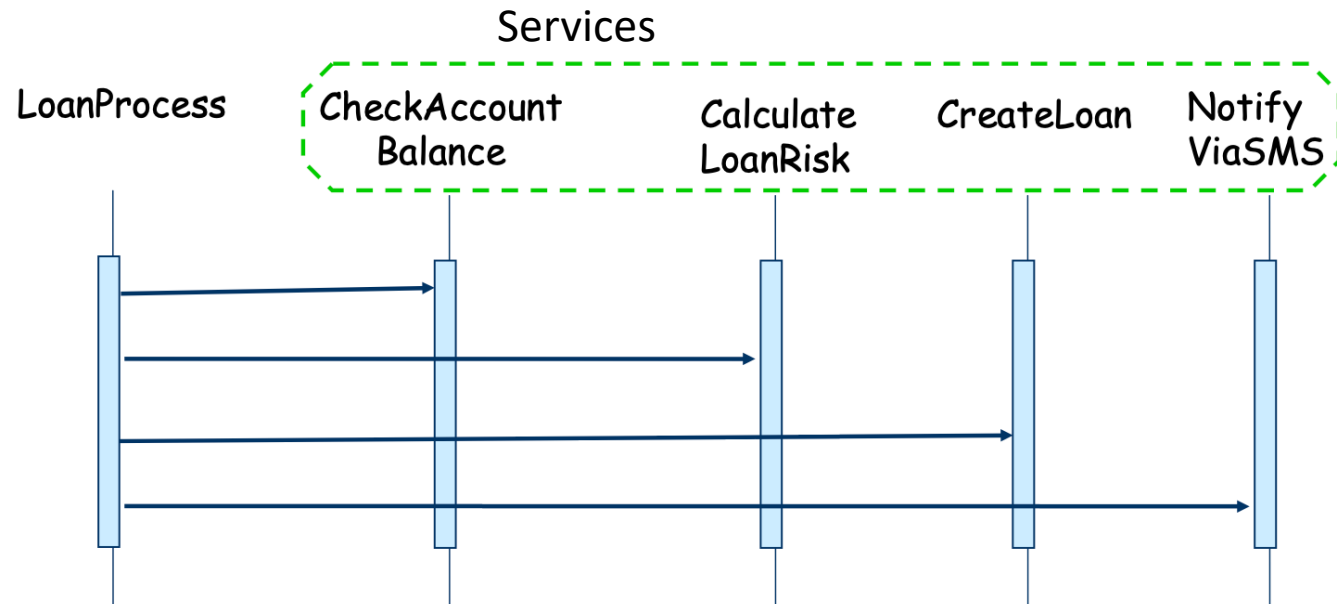
- Deux approches architecturales :
  - Les **chorégraphies** proposent une approche décentralisée dans laquelle il n'y a pas de coordinateur central. Elles supposent que les services sont capables de s'organiser pour communiquer les uns avec les autres.
  - **L'orchestration** repose sur une entité centralisée effectuant tous les appels de méthodes sur les services de l'application et relayant les messages entre les services. Les services qui composent une orchestration n'ont pas conscience d'en faire parti.

# Orchestration

E.g., LoanAgent est un assemblage de composants



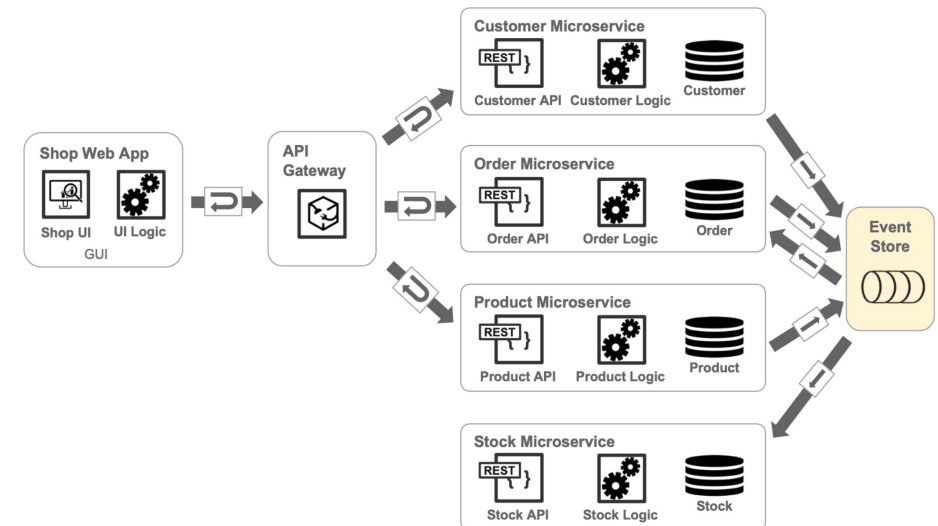
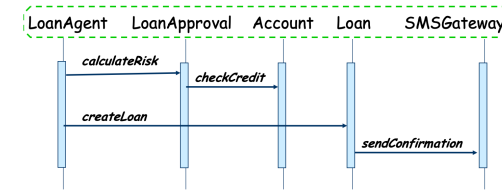
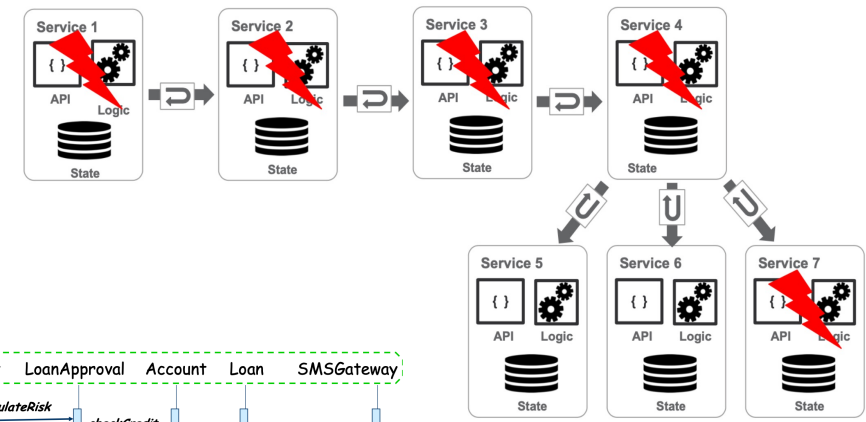
E.g., LoanAgent est un processus métier  
Une orchestration de services !  
Attention : LoanAgent peut vite devenir  
Point central de logique et de failure !





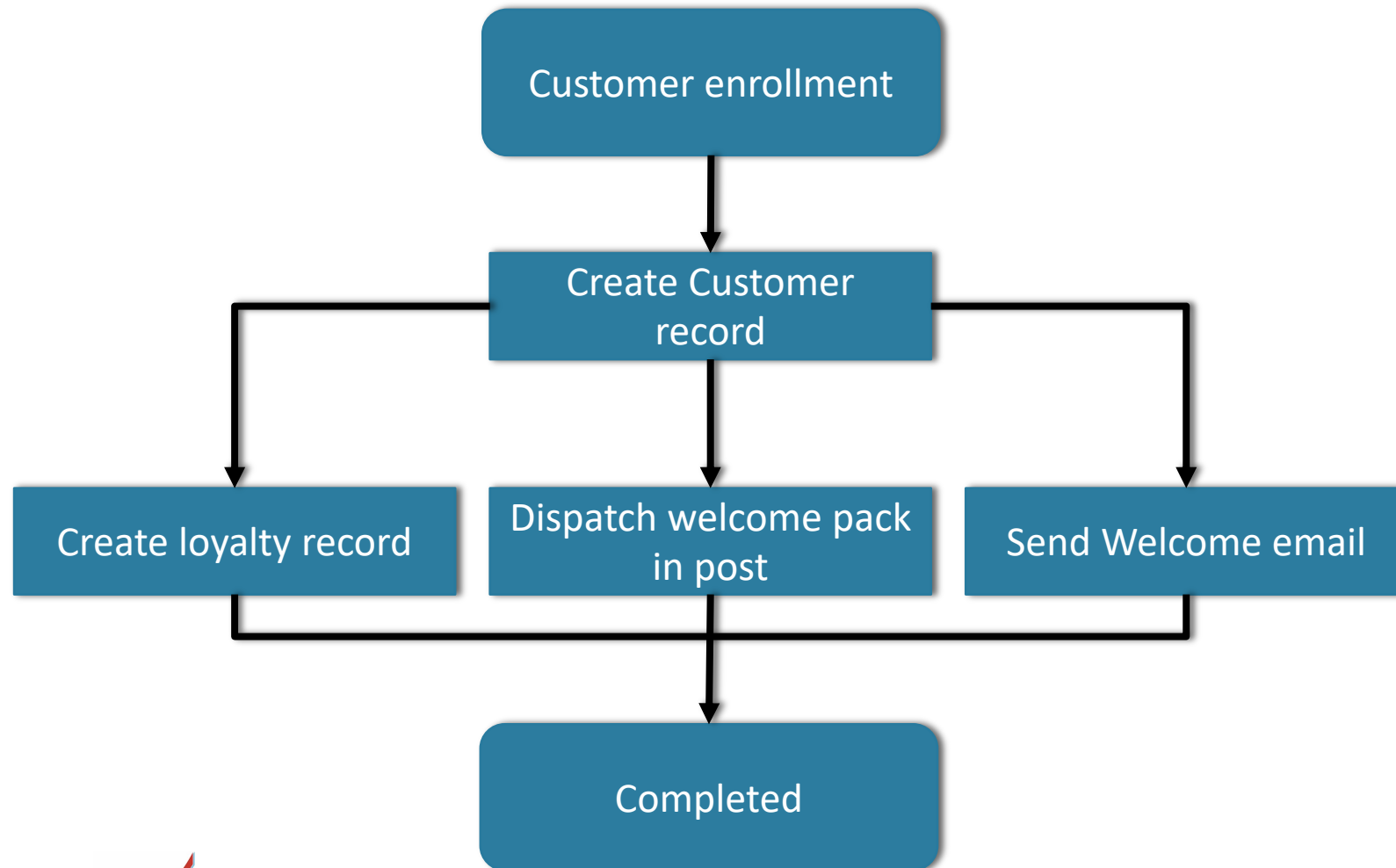
# Chorégraphie

- Attention à ne pas tomber dans le piège du couplage fort entre les services !
- On peut continuer à garantir un couplage faible. Par exemple en couplant Service et EDA !
  - Un service réagit à un événement !



# Exemple

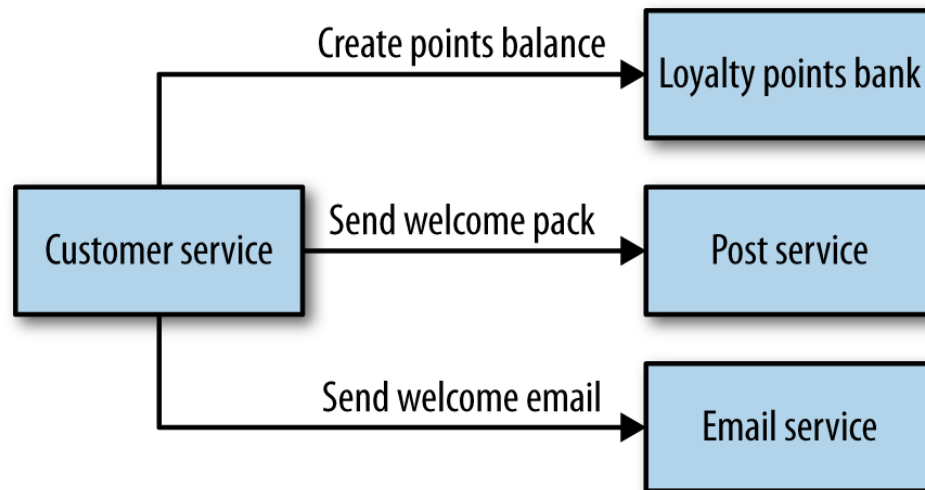
---



# Orchestration VS Chorégraphie

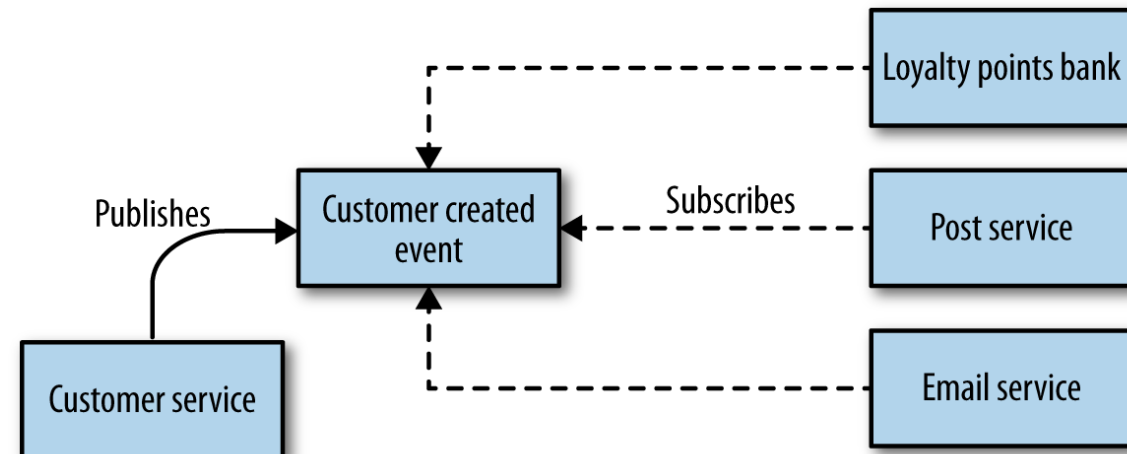
## Orchestration

- Attention SPOF / SPOC

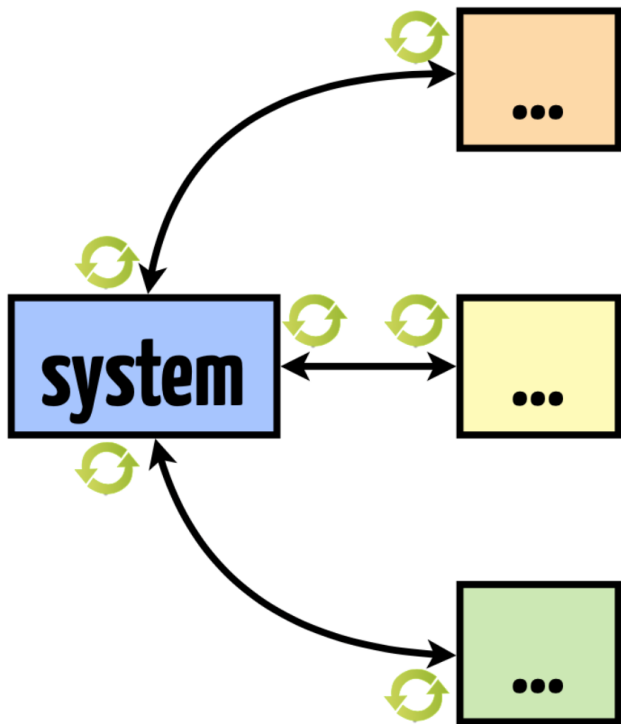


## Chorégraphie

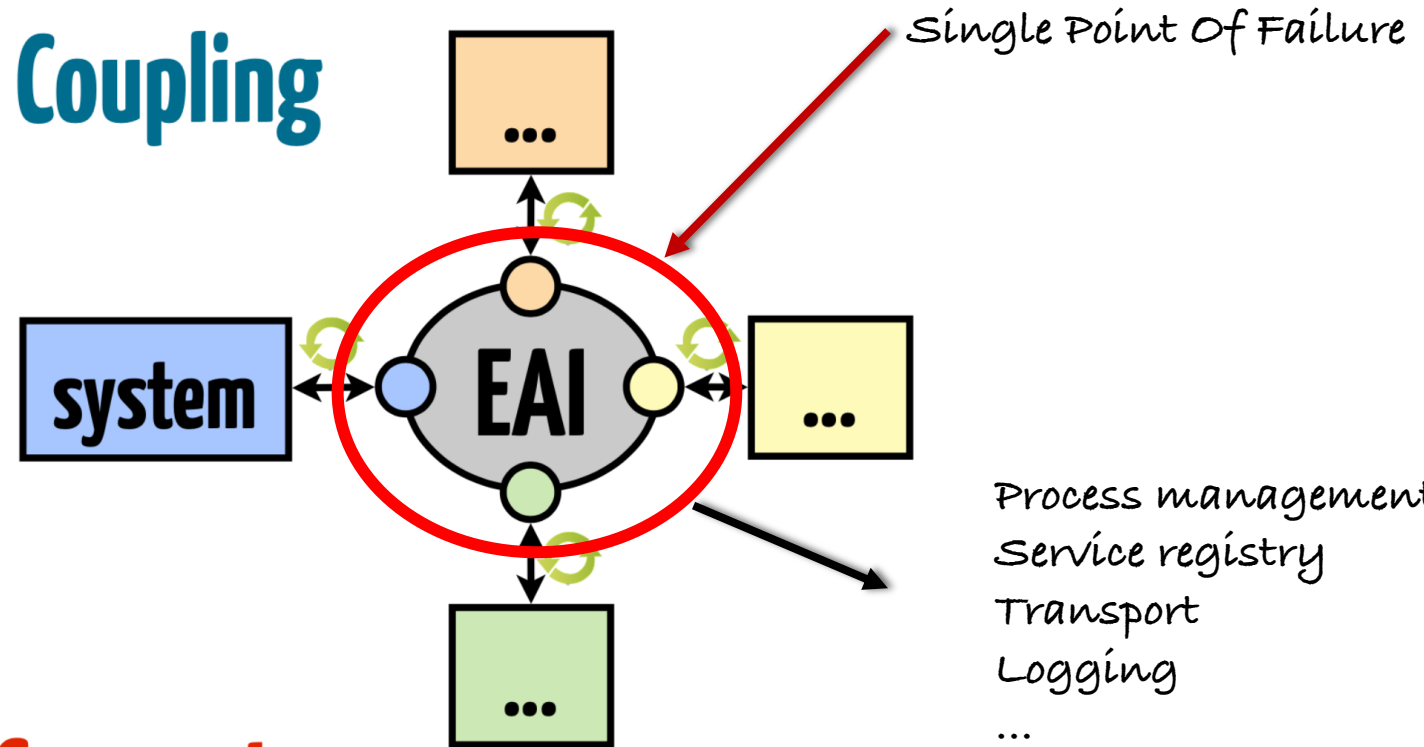
- Meilleur découplage.
- Plus complexe à tester et mettre en œuvre.
- Le process n'est plus explicite.



# Enterprise Architecture Integration

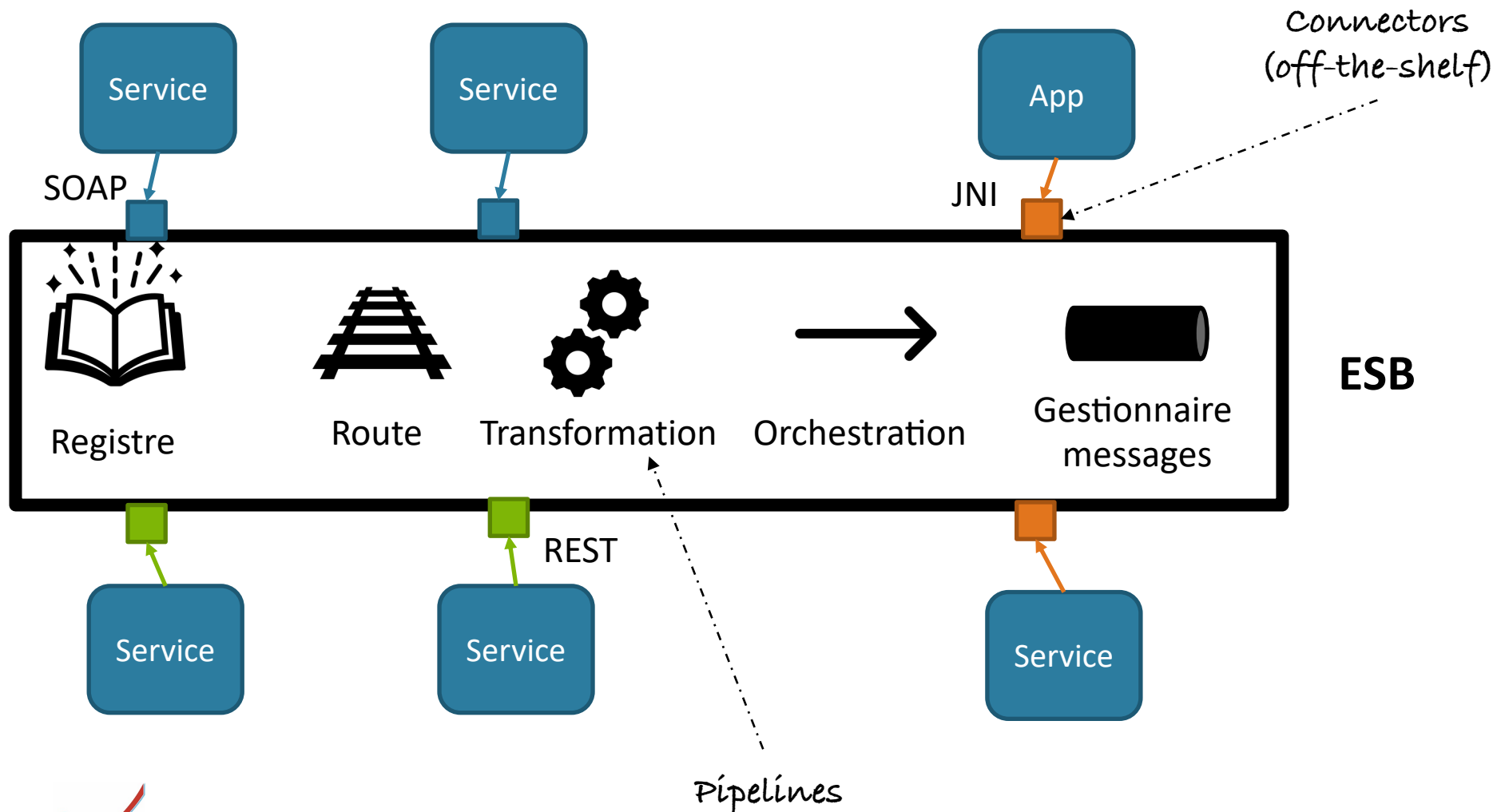


**Coupling**



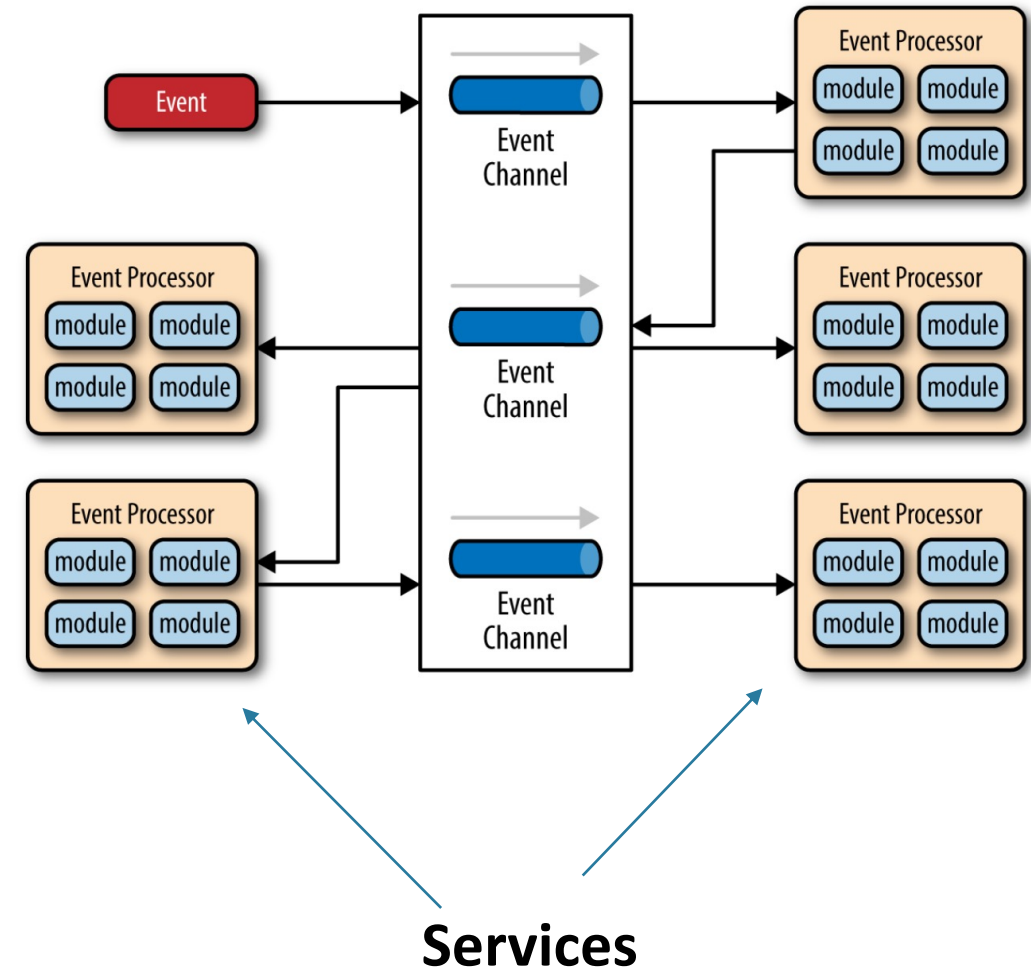
**Connectors**

# Enterprise Service Bus (ESB)



# Retour sur les EDAs

- Les deux vont très bien ensemble.
  - Combiner le meilleur des deux mondes.
- Dans les faits un event processor peut très bien être un service !
  - Un approche en vogue pour les microservices !



# Vers les microservices !

Un survol pour finir le cours !

# Vous avez déjà les bases !

---

- In the microservice architecture, an application consists in a collection of *loosely coupled* and *fined-grained* **services**
  - Microservices architecture are a way to implement SOA. Sometimes seen as a refinement/improvement of SOA. Somehow, the definition of services is also refined, yet « microservices » are services.

*« the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. »*

-- <https://www.martinfowler.com/articles/microservices.html>



# (Micro)Services

---

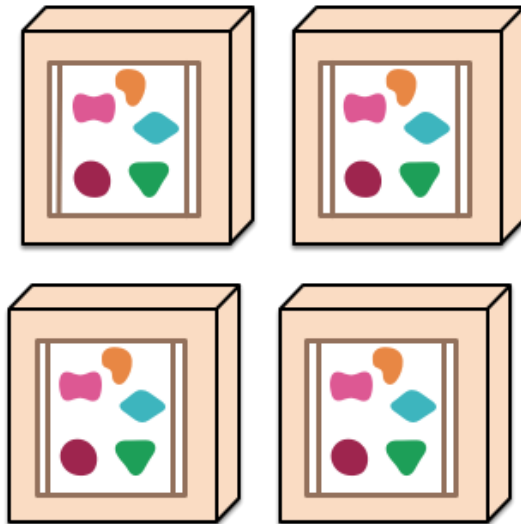
- Placer chaque **fonctionnalité** d'une application dans un service.
  - Communique via des API en général distribuées.
  - Souvent chorégraphiés via des interactions de type REST. Le moins de logique possible dans les interactions entre services.
  - Lui-même bien souvent dans un container.
- Déployables indépendamment les uns des autres.

# Monolithic vs Microservices

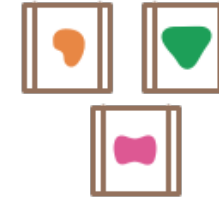
*A monolithic application puts all its functionality into a single process...*



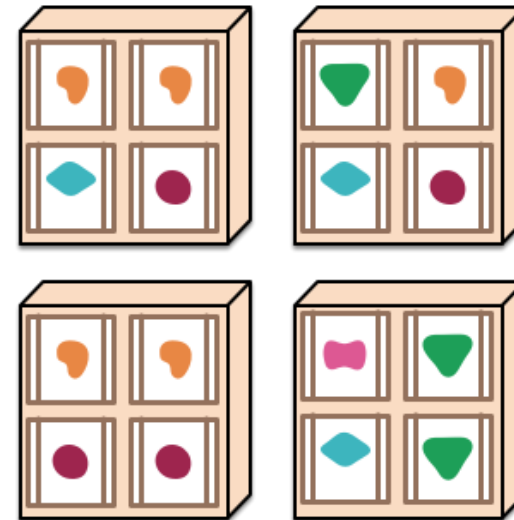
*... and scales by replicating the monolith on multiple servers*



*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by distributing these services across servers, replicating as needed.*



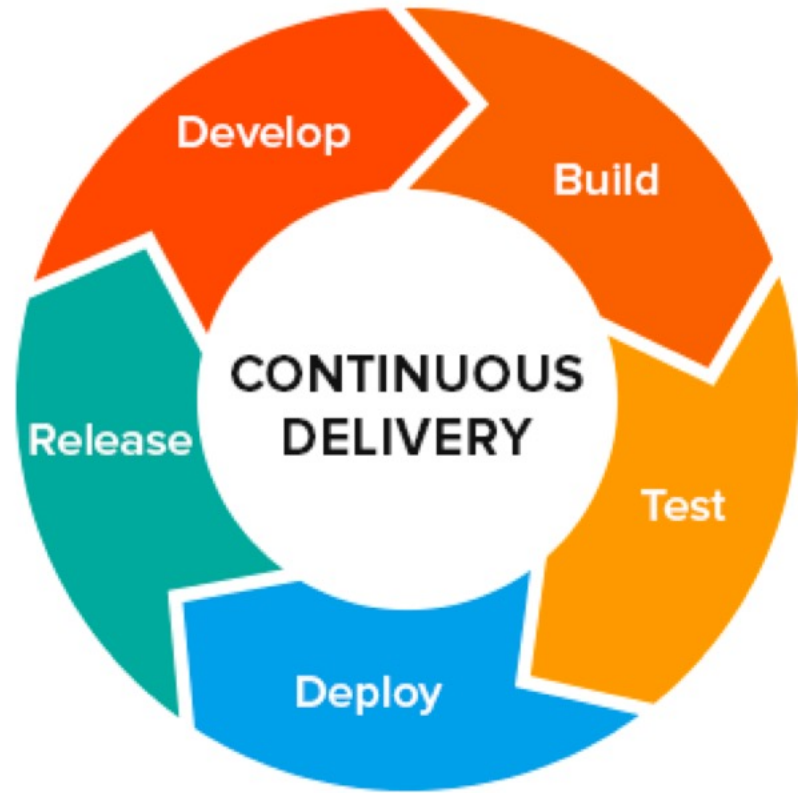
# You build, you run it !

---

- Puisque les fonctionnalités sont dans des services indépendants, on peut avoir des équipes différentes, chacune en charge de son service :
  - Du dev ... au déploiement.
    - L'équipe déploie.
    - Observe les changements.
    - Gère les erreurs (rollback ...).
- En conséquence les équipes ont des compétences multiples : user-experience, database, and project management.

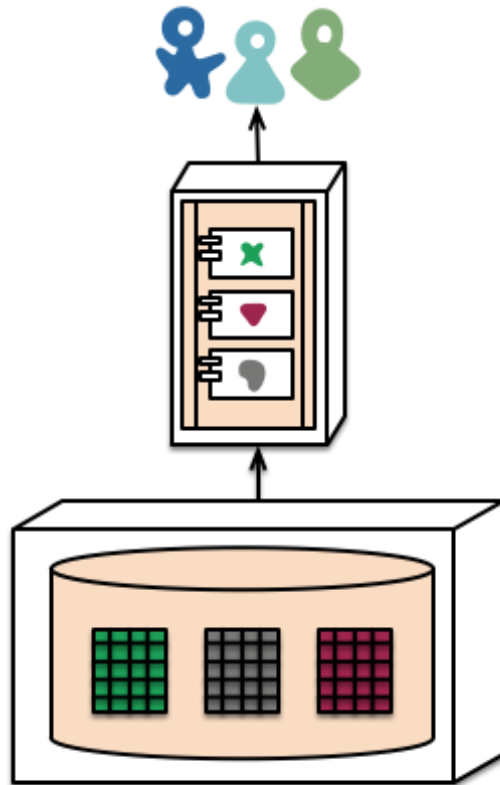
*« Amazon's notion of 'built it, run it' where a development team takes full responsibility for the software in production. »*

# Démultiplier le déploiement

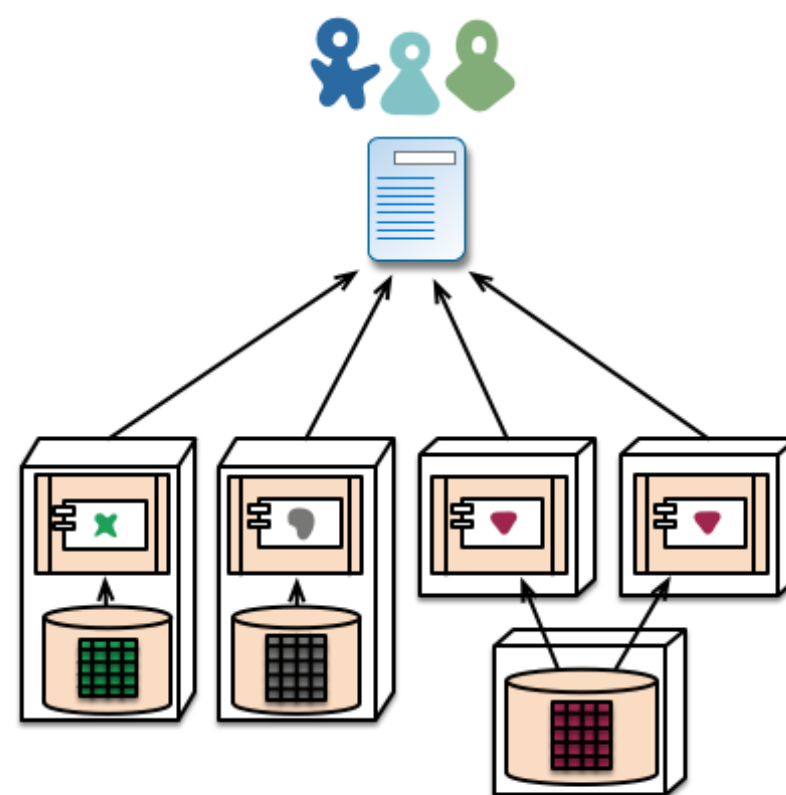


# Quelques caractéristiques des services :

## Decentralized data management



monolith - single database



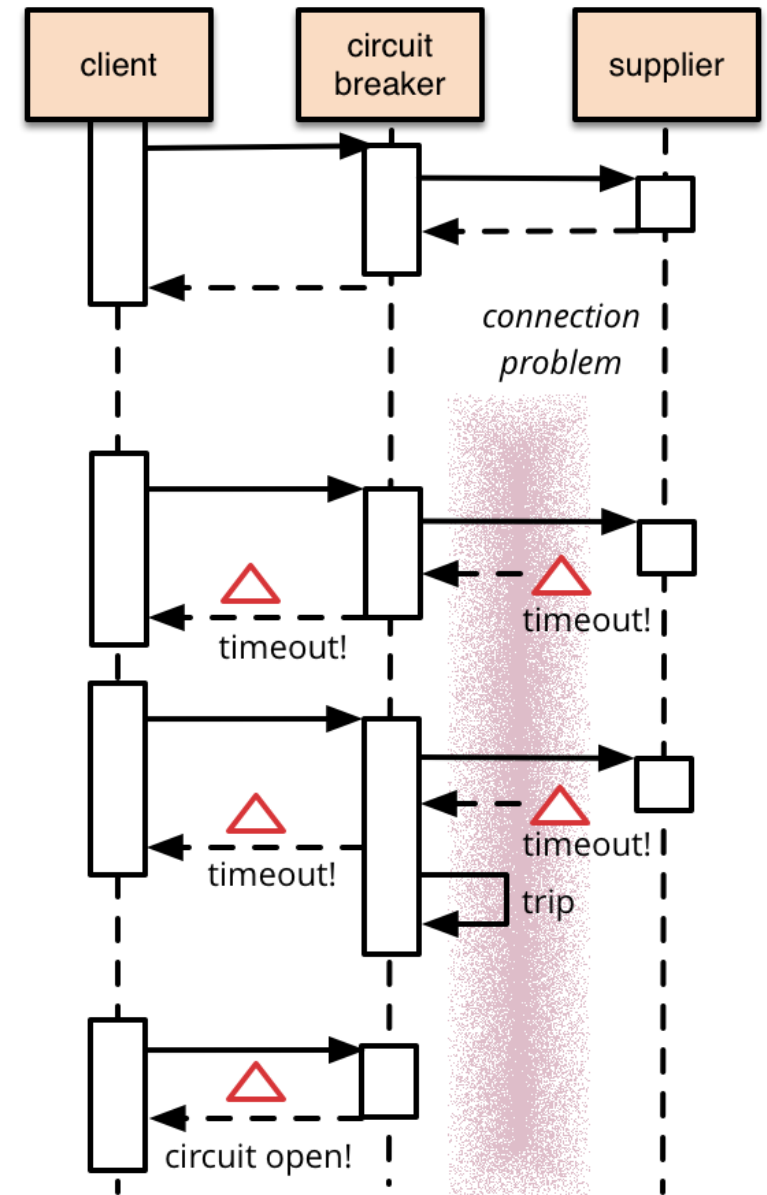
microservices - application databases

# Quelques caractéristiques des services : *Decentralized Governances (of the code)*

- Produire des outils que les autres développeurs peuvent utiliser pour résoudre des problèmes similaires.
  - Y compris du logiciel déjà largement éprouvé.
- Les équipes sont responsables de leur logiciel 24/7, y compris en production. (Extrême)
  - With great powers comes great responsibilities

# Quelques caractéristiques des services : *Design for failures*

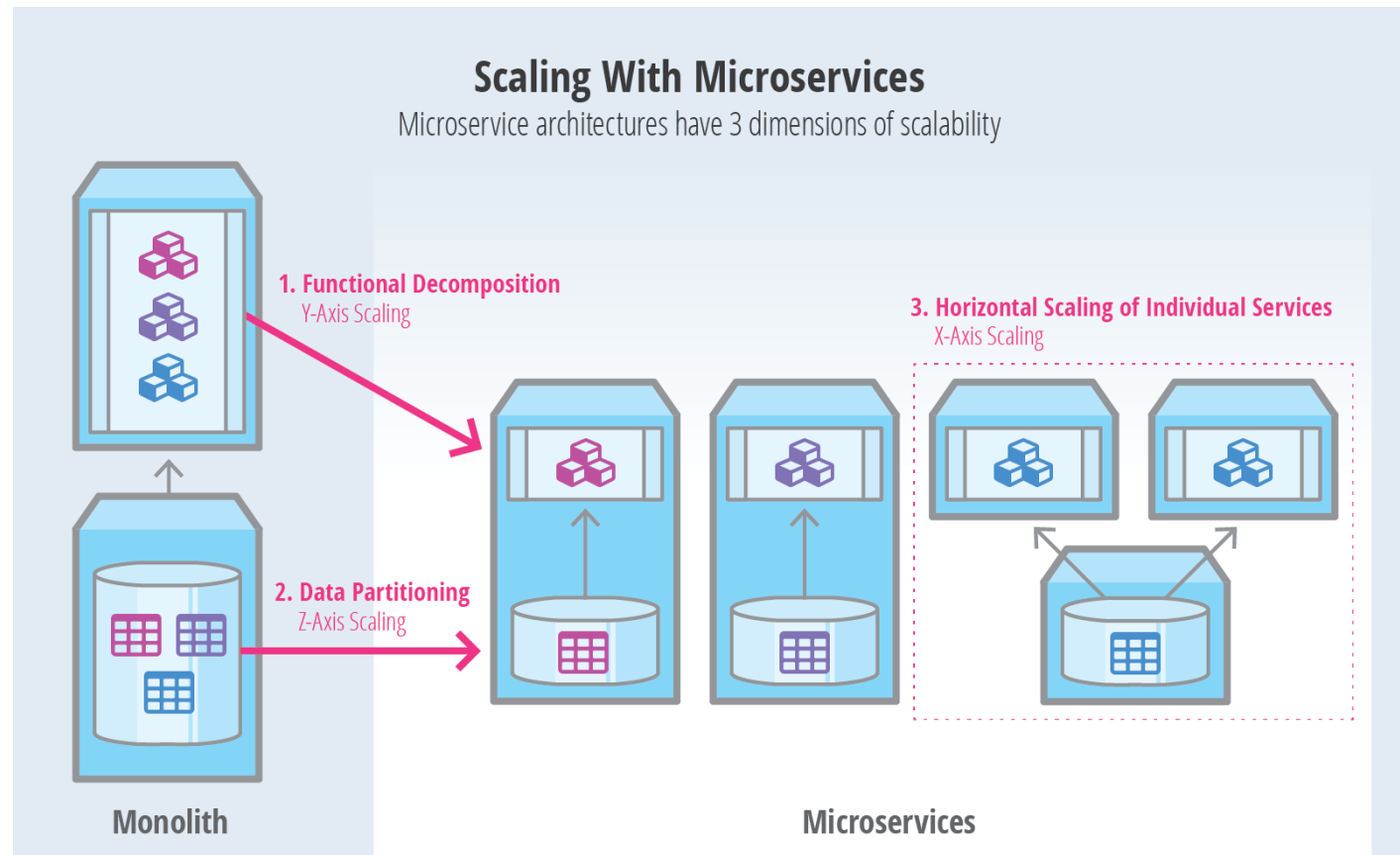
- **Monitor** : pour identifier et répondre aux problèmes le plus rapidement possible. Si possible de manière automatique.
- **Chaos Testing** : tester les services mais pas seulement, tester également la capacité du système à absorber la failure d'un/de service.
- **Des patrons** :
  - Circuit breaker : Utilisé pour empêcher des requêtes à un service en échec.



# Quelques caractéristiques des services :

## *Autoscaling*

- Micro services share no external dependencies, scaling a particular micro service instance in the flow is greatly simplified





# Summary

---

- Retour sur les points clés des services dans les architectures microservices.
  - Petits
  - Développés et maintenu indépendamment les uns des autres. Une équipe pour gérer le services de A à Z.
    - Déployables indépendamment les un des autres
  - Décentralisés
  - Interopérabilité au niveau communication
  - API + contrats
  - Construit et livré automatiquement
  - Souvent orientés message.

# Pro and cons

---

## Avantages

- Focus on delivering features to users, rather than on projects
- Decentralized governance of the code
- Independently deployable, facilitate continuous deployment and delivery
- Allow for horizontal scalability of individual services
- Design for failures and hence increase resilience
- Well-defined interfaces for low coupling
- Language independant via protocol driven interoperability

## Inconvénients

- Increase deployment and operation complexity
- Make debugging and log analysis very difficult
- Complex semantical interdependencies in case of failures
- Challenge to global data consistency

# Annexes

# JavaScript in a nutshell



# Types

---

- Typage est **faible** et **dynamique**:
  - Pas nécessaire de déclarer le type d'une variable avant de l'utiliser.
  - Le type de la variable est déterminé à l'exécution.
- **Types primitifs**:
  - Numbers
  - Symbol (donnée unique et interchangeable)
  - String
  - Boolean
  - Null
  - undefined
  - le reste est objets (y compris les tableaux)

# Bases

---

- Les instructions (statements) sont séparées par des point-virgules.

```
console.log("un test"); // Pour afficher un message sur la console
```

- Commentaires (comme la plupart des langages).

```
// un commentaire sur une ligne
```

```
/* un commentaire plus long
```

```
sur plusieurs lignes */
```

# Variables

---

- **Trois types de déclaration:**

- **var** : si dans une fonction, disponible que dans la fonction. Sinon globale.
- **let** : variables ayant le bloc courant comme portée.
- **const** : constante accessible en lecture uniquement (bloc courant comme portée).

- Une variable sans valeur initiale vaut `undefined`.

```
If(toto.a === undefined){  
  console.log("oups");  
}
```

- L'opérateur `||` peut être utilisé pour affecter des valeurs par défaut.

```
var status = flight.status || "unknown";
```

# Variables globales

---

- Les variables globales sont des propriétés de l'objet global.
- Dans le navigateur il s'agit de l'objet `window`:

```
window.istruer=true;
```



# Conditions

---

- **If**

```
if (containment !== undefined) {  
    var port_host = containment.src;  
    var host_name = that.get_comp_name_from_port_id(port_host);  
    return that.find_node_named(host_name);  
} else {  
    return null;  
}
```

- **Switch**

```
switch (new Date().getDay()) {  
    case 4:  
    case 5:  
        text = "Soon it is Weekend";  
        break;  
    case 0:  
    case 6:  
        text = "It is Weekend";  
        break;  
    default:  
        text = "Looking forward to the Weekend";  
}
```

# L'égalité

---

- Egalité faible `==`
  - Compare deux valeurs **après** les avoir converties en valeurs d'un même type.

```
var num = 0;  
var obj = new String("0");  
console.log(num == obj); //true
```

- Egalité stricte `===`
  - **Aucune des valeurs n'est convertie** avant que la comparaison soit effectuée
  - Une valeur est uniquement égale à elle-même.

```
var num = 0;  
var obj = new String("0");  
console.log(num === obj); //false
```

# Boucles

---

- **for ...**
- **for ... in**
- **for ... of (sur les objets itérables)**

```
for(let i = 0; i < 10; i++){  
    console.log("I vaut: "+i);  
}  
let arr = [3, 5, 7];  
arr.toto = "coucou";
```

```
for (let i in arr) {  
    console.log(i); // affiche 0, 1, 2, "toto »  
}
```

```
for (let i of arr) {  
    console.log(i); // affiche 3, 5, 7  
}
```

# Boucles

---

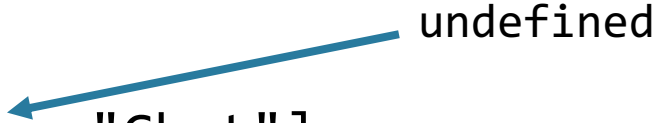
- **for ...**
- **for ... in**
- **for ... of (sur les objets itérables)**
- **while**

```
for(let i = 0; i < 10; i++){  
    console.log("I vaut: "+i);  
}
```

```
let i=0;  
while (i<10){  
    console.log(i);  
    i++;  
}
```

# Littéraux

---

- Aussi appelés initialiseurs d'objets (cf. transparents suivant).
- Représentent des valeurs fournies littéralement.
- Tableaux: `var poisson = ["Clown", , "Chat"];`  

- Booléens: `var poisson = true;`
- Strings: `var poisson = "poisson";` ou `var poisson = 'poisson';`

# Objets

---

- **Les objets sont des conteneurs de propriétés.**
  - Une propriété a un nom (string) et une valeur (tout sauf undefined).
  - Un objet peut contenir d'autres objets.
- **Un objet peut hériter des propriétés d'un autre objet.**
  - Pas de concept de classe mais des prototypes.
  - Prototype: modèle fournissant des propriétés initiales pour un nouvel objet.
  - Un objet peut devenir le prototype d'un autre.

# Créer un objet

---

- **Via un littéral:**

- Prend la forme de paires clé/valeur entre des accolades.

```
var empty_object = {};
```

```
var something={  
  "something1": "cool",  
  "something2": "not so cool"  
  "something3": { // an object  
    "a_final_one": "done"  
  }  
};
```

# Créer plusieurs objets du même type ?

---

```
var Jean={
  nom: "jean",
  age: 50,
  points: 20,
  updatePoints: function(pts){
    this.points -= pts;
  }
};
```

```
var Pierre={
  nom: "Pierre",
  age: 45,
  points: 20,
  updatePoints: function(pts){
    this.points -= pts;
  }
};
```

**On ne veut pas dupliquer ce genre de code !**



# Créer un objet

- **Via un constructeur (i.e., une fonction de création) :**

// Définition de l'objet Person via un constructeur

```
function Person(nick, age, sex, parent, work, points) {  
  this.nick = nick;  
  this.age = age;  
  this.sex = sex;  
  this.parent = parent;  
  this.work = work;  
  this.points = points  
  updatePoints = function(pts){  
    this.points -= pts;  
  }  
}
```

*Convention de nommage, première lettre en majuscule*

// On crée des variables qui vont contenir une instance de l'objet Person:

```
var seb = new Person('Sébastien', 23, 'm', 'aîné', 'JavaScripteur', 20);  
var lau = new Person('Laurence', 19, 'f', 'soeur', 'Sous-officier', 30);
```

# Accéder aux propriétés d'un objet

---

- **Deux manières:**
  - String entre crochets : `flight["status"]`
  - Avec la notation "." : `flight.status`
- **Egalement pour la mise à jour:**
  - `flight.status = "unknown"`
  - `flight["status"] = "unknown"`
  - **Si la propriété n'existe pas, l'objet est augmenté :**  
`flight.new_property = "new";`
- **Les objets sont passés par référence (et non pas copiés).**
  - `var x = flight;`
  - `x.test = "a test";`
  - `var resultat_test = flight.test; //resultat_test vaut "a test"`

# Fonctions

---

- Une fonction est un objet, utilisable comme n'importe quel objet :
  - Peut être stocké dans des variables, objects, arrays.
  - Peut être passé en paramètre à une fonction.
  - Peut être retourné par une fonction.
  - Une fonction peut contenir une fonction.
- Contient un ensemble de **statements / instructions**.
- Vient avec un **contexte**.

# Fonctions - Literal

```
var unObjet = {  
  value: 0  
  increment: function () {  
    this.value ++;  
  };  
}
```

```
unObjet.increment();
```

```
var add = function addition(a,b) {  
  if(a === 1 && b !== 1){  
    return addition(b, b); // I can call addition  
  }else{  
    return a+b;  
  }  
};
```

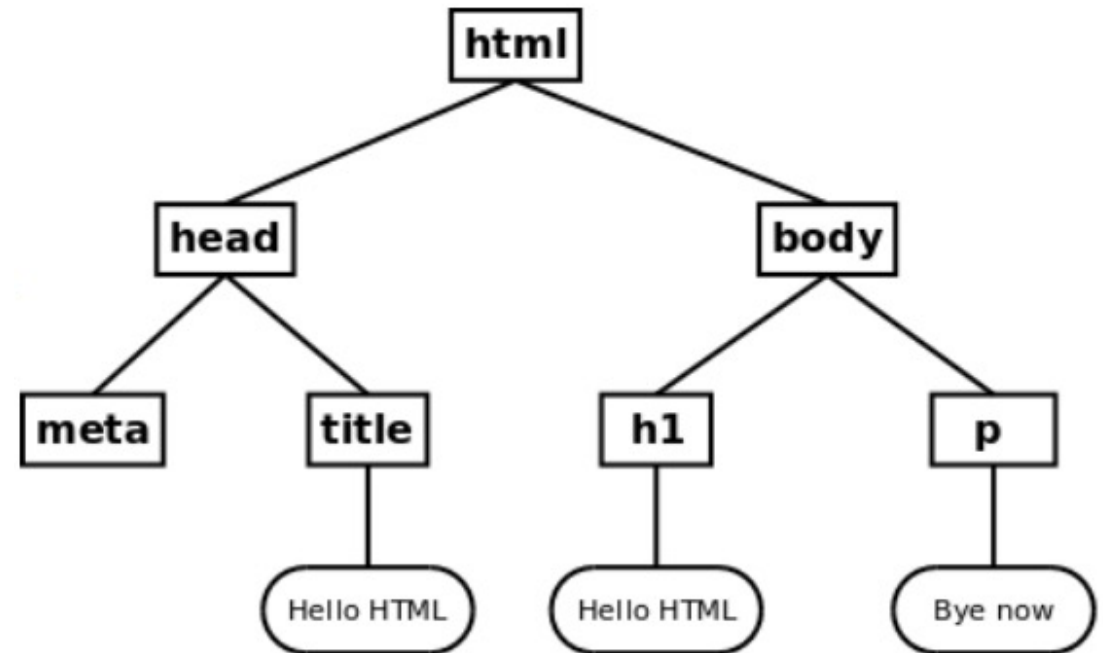
```
var sum = add(1,3); // sum vaut 6
```

# Document Object Model (DOM)

- Comment modifier la page web en fonction des actions de l'internaute ?
  - Modifier la page c'est modifier ces noeuds
  - Il faut un **modèle** de la page et de sa structure, ainsi qu'une API pour modifier ce modèle.

## ➤ DOM

- 1998 : Standardisation par le W3C du DOM niveau 1



# DOM

---

- Le navigateur « **parse** » le **HTML** pour **construire l'arbre DOM en mémoire** et **construit une représentation de cet arbre en suivant les règles de style CSS**.
- Javascript implémente l'API DOM pour manipuler dynamiquement les pages web
  - Une modification du DOM est une **modification du modèle de la page dans la mémoire** du navigateur.

# Les objets du DOM

---

- **Document** : l'élément racine à partir duquel est construit le DOM
- **Node** : Les noeuds de l'arbre, peuvent être **Element** (un élément HTML) ou **CharacterData** (contient du texte)
- **Event** : Les événements
- **NodeList** : Liste de noeuds

Plus ici: [https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)

# Sélection d'éléments

- La sélection d'éléments peut se faire via:

- L'identifiant de l'élément (attribut `id`)

```
var element = document.getElementById("unique");
```

- Le nom de l'élément (attribut `name` pour certains éléments seulement)

```
var element = document.getElementsByName("a_name");
```

- La classe de style de l'élément (attribut `class`)

```
var element = document.getElementsByClassName("important");
```

- L'élément lui-même

```
var element = document.getElementsByTagName("div");
```

- Un sélecteur CSS

```
var listElement = document.querySelectorAll("div.exercice ul");
```

Retourne  
une liste



# Manipulation des propriétés

- Pour un Element il est possible de changer :
  - Ses attributs.
  - Son contenu.
  - Son style CSS.
- Les attributs des éléments HTML deviennent des propriétés de l'objet DOM.

```
var monImage = document.getElementById("portrait");  
var taille = monImage.width;  
monImage.className = "portrait";
```

L'attribut classe devient className

# Manipuler le contenu et le style

---

- On peut modifier le contenu :
  - HTML d'un élément  
`var htmlText = element.innerHTML;`
  - Textuel d'un élément  
`var txt = element.textContent;`
- Et le style:
  - Via la propriété `style`  
`element.style.fontSize = "12px";`

# Événements

---

- Certaines actions sur des éléments produisent un **événement**.
- Différents types d'événements:
  - Action utilisateur : click, keypress, mouseover, ...
  - Changement d'état : focus, change, ...
  - Chargement : load

Sounds like a callback ...

`objet.addListener(eventType, listenerFunction)`

Viennent avec leurs propriétés

# Evénements

---

- Certaines actions sur des éléments produisent un **événement**.

Sinon on peut faire par exemple:

```
<button onclick="envoyerCommande()"></button>
```

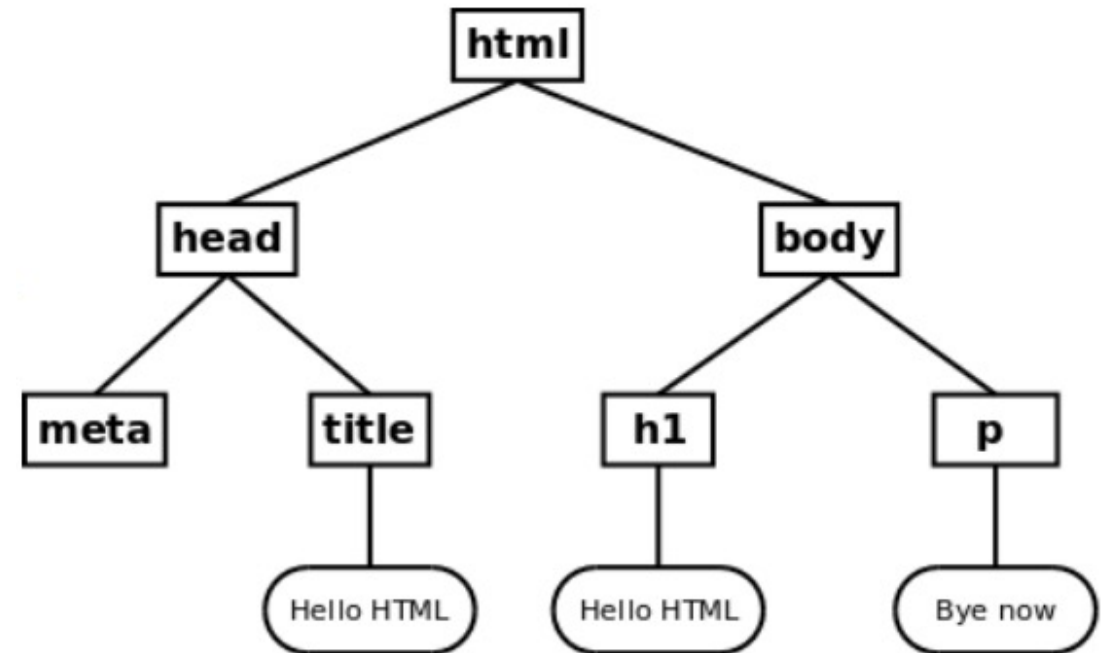
back ...

Viennent avec leurs propriétés



# Modifier l'arbre

- Il est possible de :
  - Créer un Element  
`document.createElement();`
  - Créer un noeud de texte  
`document.createTextNode();`
  - Cloner un noeud  
`noeud.cloneNode();`
  - Insérer un noeud  
`parent.appendChild(noeud);`
  - Supprimer un noeud  
`parent.removeChild(noeud);`



# Requête HTTP

- Désormais supporté par la plupart des navigateurs.

```
fetch("https://www.une-url.com")  
.then(response => response.json())  
.then(response => alert(JSON.stringify(response)))  
.catch(error => alert("Erreur : " + error));
```

Par défaut GET, l'url que l'on souhaite interroger

Lorsque la réponse est obtenue (promesse) on transforme en JSON

Lorsque la transformation en JSON est terminée (promesse) On fait qqchse avec notre réponse !