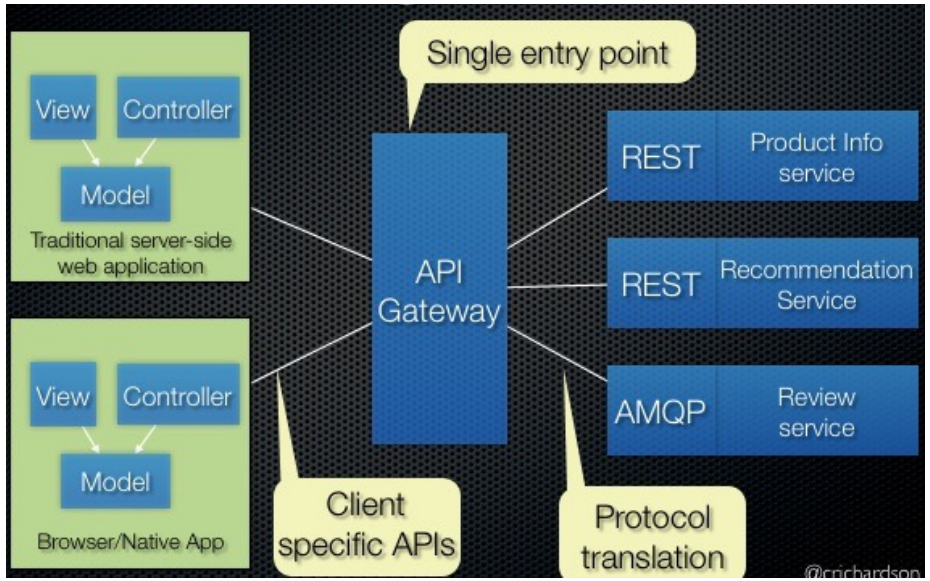
IUT Lyon 1
l'excellence technologique

Chapitre 7 – Patterns d'Architecture



Véronique DESLANDRES - LP DevOps, IUT Univ. Lyon1

Module Conception d'Architectures Logicielles

Sommaire de ce cours

© V. Deslandres, IUT Lyon1

Introduction

Pattern de **Conception Logicielle** (Design Pattern)

- Lors de la conception d'un logiciel, surtout Orienté Objet
- Indépendant du langage et des technologies

Pattern **d'architectures**

- Certains généraux : **multi-niveaux, client serveur, MVC**, **MVVM, DAO**
- D'autres plus spécifiques : **API Gateway**, **Event sourcing**, etc. un grand nombre orientés **architecture microservice**
- **Nomenclature non unifiée !**
- Indépendant des plateformes technologiques

Anti-patterns : définition

Anti-patterns : mauvais choix de conception, par exemple :

- Architecture 3-tiers !
  - Pour les utilisateurs de microservices, l'archi 3-tiers conduit à des applications monolithiques
- Le **verrouillage à double test** d'accès concurrent
  - Aussi appelé ***Double-checked locking***
  - Les solutions proposées (comme le *synchronized* sur la *méthode getInstance()* ou sur le bloc d'instanciation) sont en faites dépendantes de la machine utilisée : on ne sait pas ce qui va se passer réllement
- Mais aussi le non respect des principes de COO : instabilité du code (parties *variable* et *stable* pas séparées), trop grande complexité

Patterns d'architecture logicielle

Objectif

- Définir la meilleure architecture de son application en fonction des *technologies utilisées*, des *contraintes techniques* mais aussi *des usages envisagés*
- Pour avoir une bonne *efficacité de l'application,*
- *Une bonne productivité* du développement et de maintenance de l'application.

Différents types d'architecture

- **Patrons multi-niveaux** (*layered pattern*)
  - Un niveau : une responsabilité
  - Niveaux indépendants
  - Ex. : certains sites d'eCommerce
- **Patrons Client-Server** : un serveur, de multiples clients
  - Ex.: applications de messagerie, du web, de partage de fichiers, les applications des banques, des médias, etc.
  - Souvent des **RESTful** applications (*Representational State Transfer*)

© V. Deslandres, IUT Lyon1

Patterns d’architecture logicielle (2)

- **Event-Driven Pattern** : opérations tirées par les événements
  - Exemple : inscription à un service, le fait de valider ses informations sur le formulaire crée un compte
    - Valider une inscription (événement) → Création de compte (action)
  - C’est le cas des applications Client riche, la plupart des sites d’eCommerce

- Patterns **d’Architecture Microservices**
  - Détaillés ci-après

Patterns Microservice : typologie

L'architecture MS est une architecture de développement qui permet de déléguer des fonctionnalités à des **systèmes tiers**.

  - Ces derniers sont très nombreux : gestion des identités, authentification, paiement, cartographie, envoi de SMS, etc.

- Ces services partagent un certain nombre de notions, telle la communication, la gestion des données, etc.

Différents regroupements des patterns MS existent dans la littérature. Dans ce cours, nous considèrerons :

- **Patterns d'intégration**
- **Patterns de communication**, notamment asynchrone
- **Patterns de Gestion des Données (CQRS, Event Sourcing, Saga, etc.)**

© V. Deslandres, IUT Lyon1

Patterns d'intégration

- 3 patterns d'intégration les plus utilisés dans les architectures de microservices :
  - Pattern de Composition d'APIs
  - Pattern de Passerelle d'APIs (API Gateway)
  - Pattern BFF (BackEnd for FrontEnd)

Containers de données en Architecture MS

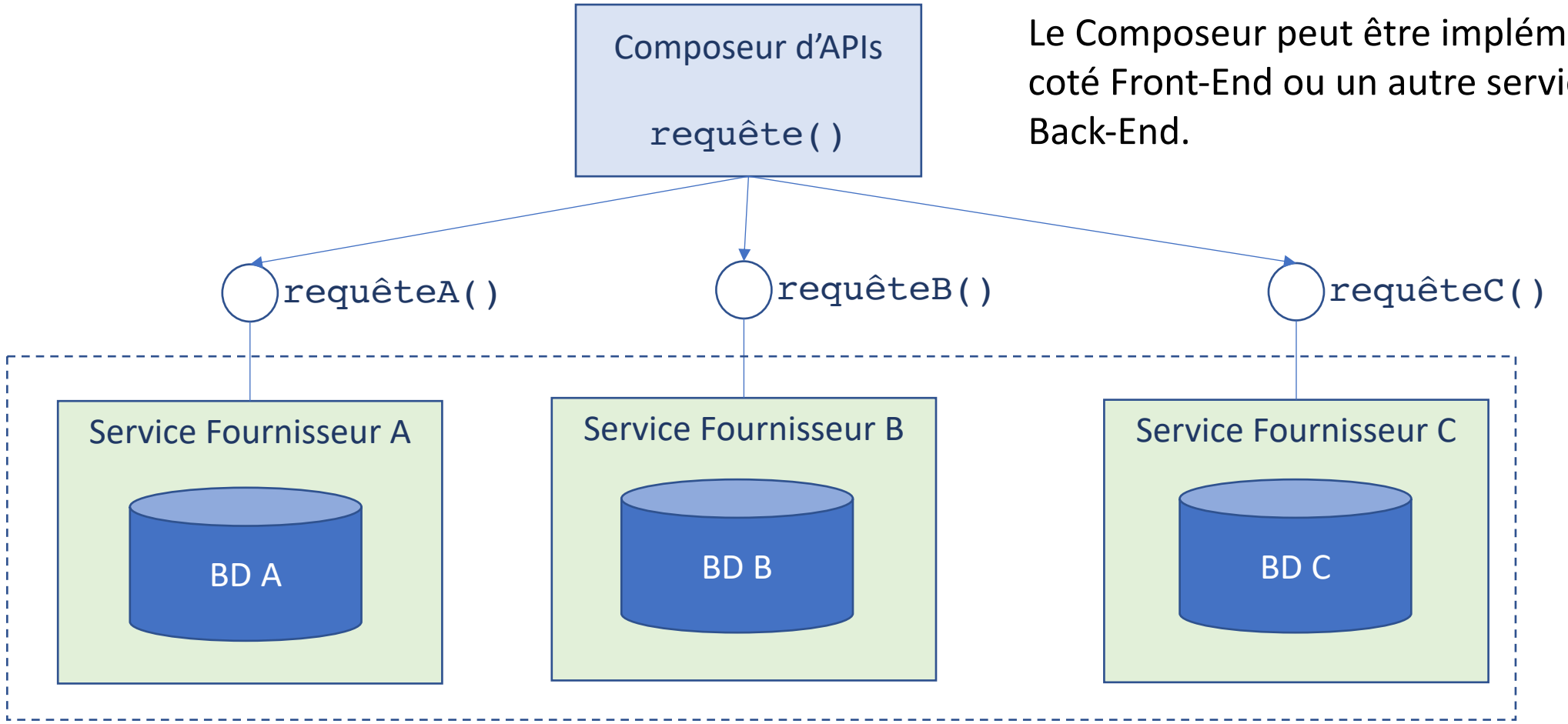
- **Problème** : comment gérer l'accès aux données quand les services gèrent des données présentes dans différents containers ?

Plusieurs solutions :

- Une Base de Données par service
  - Avantage: simple à implémenter
  - Inconvénient : problèmes de performance et synchronisation
  - Patterns concernés : Composition d'APIs, Saga, CQRS
- Une Base de Données partagées entre plusieurs services
  - Avantage: simple à implémenter (si accès concurrent)
  - Inconvénient : risque de goulet d'étranglement
- Des solutions mixtes existent

Pattern de Composition d'APIs

- Contexte
  - Une application à base de microservices, chacun ayant sa propre BD
- Problème
  - Comment implémenter des requêtes qui combinent des données provenant de plusieurs microservices, donc de plusieurs BD ?
- Solution
  - Implémenter cette requête avec un *composeur d'APIs*.
  - On ajoute un composant entre les MS et l'application, qui va effectuer une jointure en mémoire des résultats.



Le Composeur peut être implémenté coté Front-End ou un autre service en Back-End.

Chaque service est propriétaire de ses données

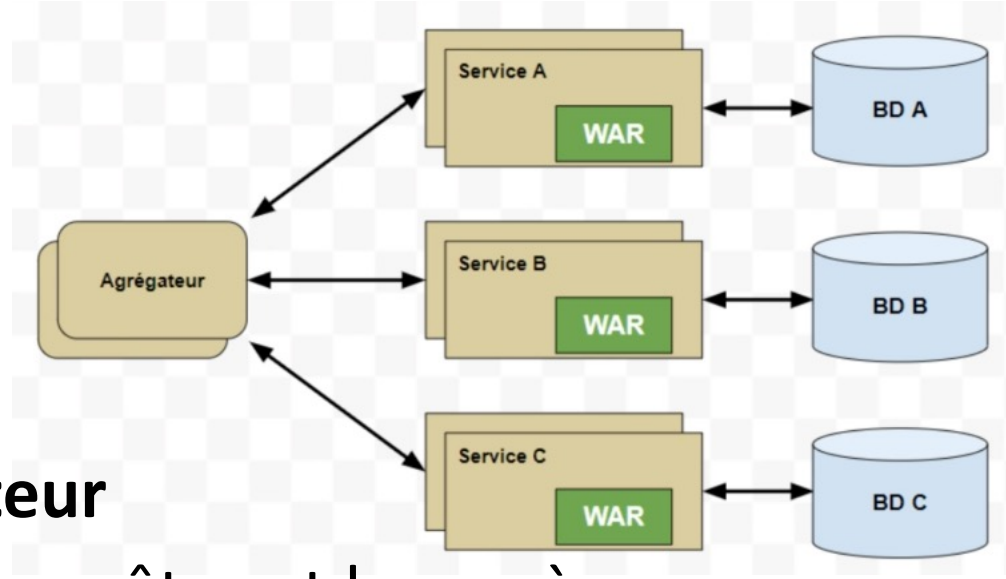
Le *composeur* implémente une requête qui invoque chaque service et combine les résultats en mémoire.

3 implémentations possibles

- Composition par agrégation
- Composition par chaînage
- Composition par branchement

Etudions chaque implémentation en détail

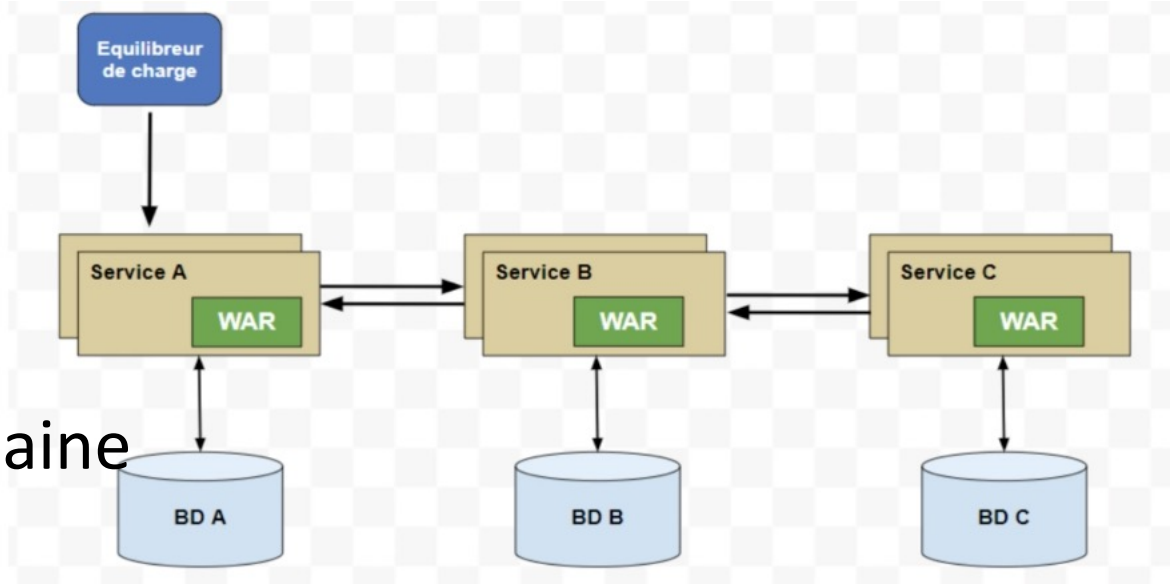
Pattern Agrégateur



- Forme la plus répandue

- Le composeur d'APIs est appelé : **Agrégateur**
- Souvent une page web, qui distribue les requêtes et les agrège
  - Chaque service est exposé par un REST léger, on applique une logique d'affaire
  - La page web récupère les données, les traite et les met en forme sur la page
- L'agrégateur peut aussi être un autre service de niveau supérieur
  - Applique une logique d'affaire
  - Peut être exploité par d'autres services ou une appli web
- On peut créer plusieurs instances possibles (des services ou de l'agrégateur), ce qui permet l'équilibrage de charge.

Composition par chaînage

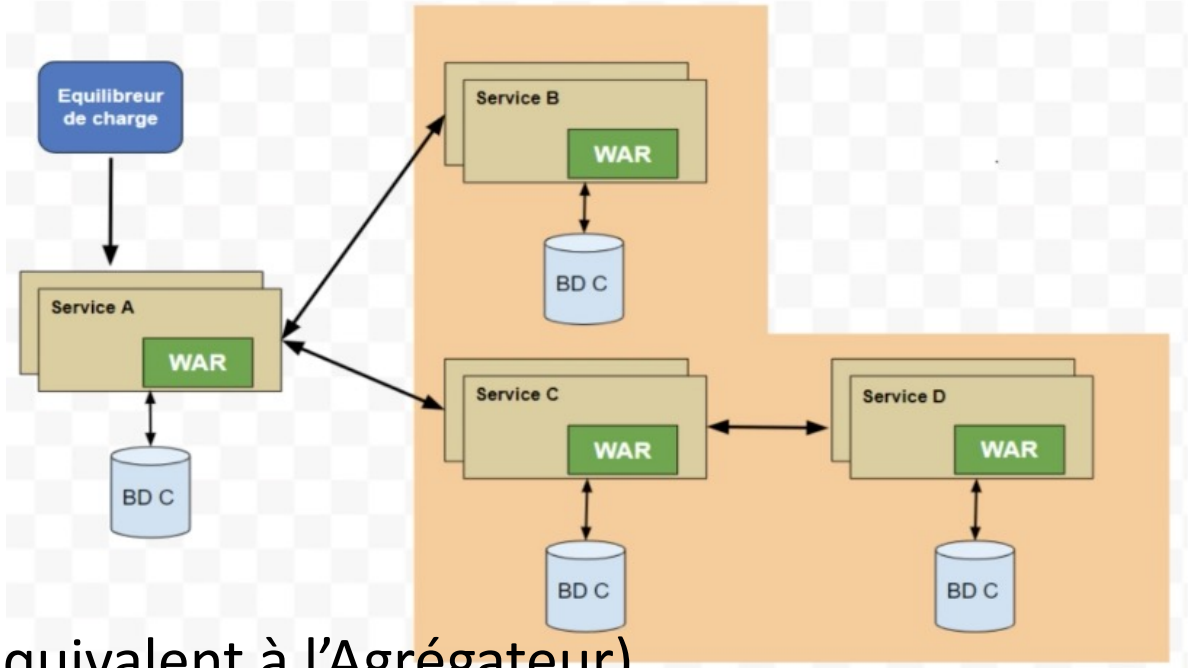


- Requêtes effectuées « en chaine »
- Chaque service ajoute sa valeur à la chaine
- Communication **synchrone** impérative
- Bloquant : le Client attend la réponse finale
  - Par ex. une page qui attend la réponse pour être affichée

- Eviter les chaines trop longues !
- A utiliser quand il y a des dépendances entre les Services
- **Chaine Singleton** : une chaine avec un seul MS
  - Avec l'idée d'ajouter d'autres MS *plus tard*

Composition par branchement

- Extension de l'agrégateur



- On a des branches qui sont appelées simultanément, ou l'une après l'autre

(Si appel simultané, par ex. ici à B et C, c'est équivalent à l'Agrégateur)

- Branches = chaines, chacune pouvant avoir un seul élément
- Différentes chaines : qui peuvent être exclusives

Pattern de Composition d'API

- Avantage
  - Simplicité : c'est un moyen simple d'interroger des données de différentes ressources dans une architecture MS

- Inconvénient
  - Certaines requêtes peuvent générer des jointures en mémoire, inefficaces pour de gros volumes de données
  - Une solution dans ce cas : la Passerelle d'API

Les patterns de communication

Asynchrones et synchrones

Modèles de Communication entre les MS

- Dans l'architecture des microservices, les applications logicielles sont conçues comme une suite de services indépendants. Ainsi, afin de réaliser un cas d'utilisation métier, il est nécessaire d'avoir les structures de communication entre les différents microservices / processus.

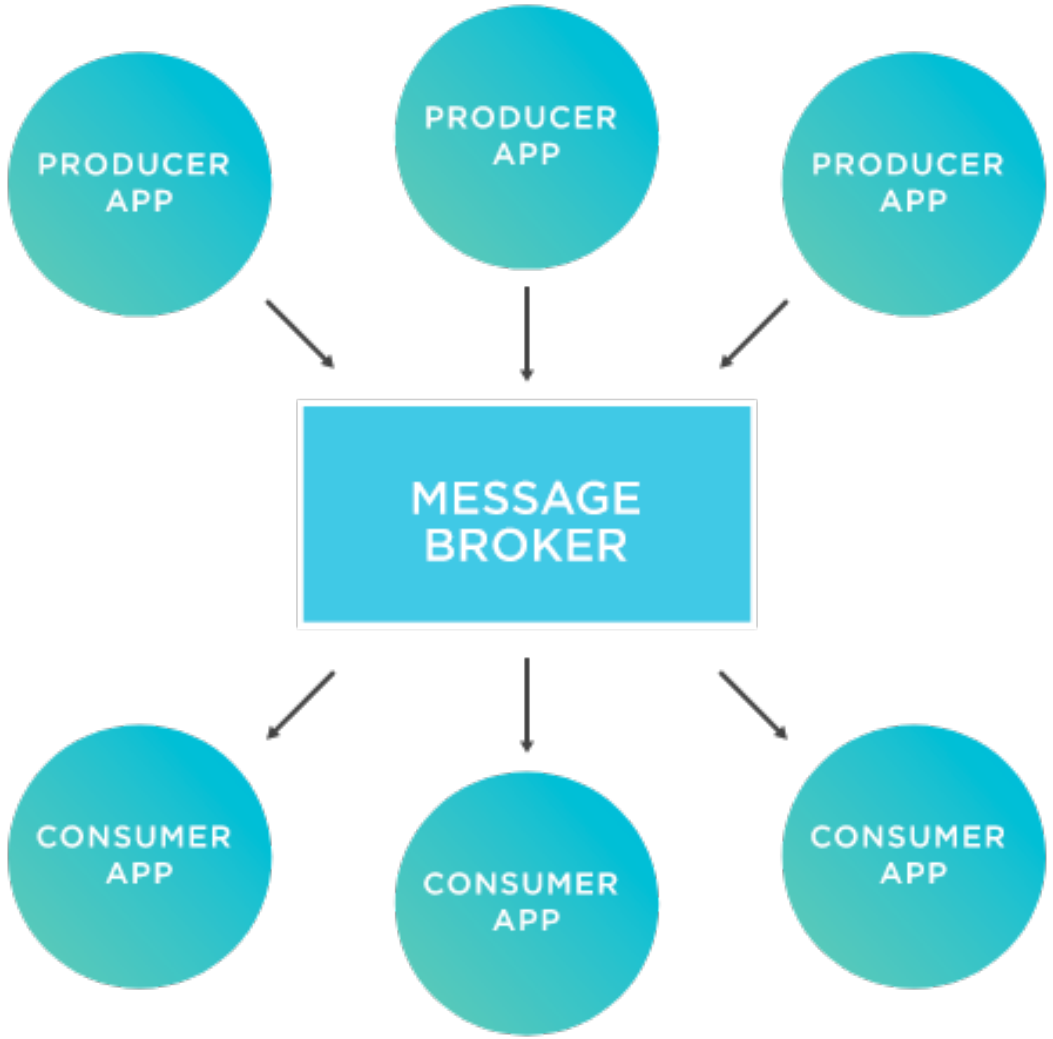
- Tandis que les composants **SOA** (Service-Oriented Architecture) communiquent entre eux à l'aide d'un ESB (Enterprise Service Bus) muni d'une logique Métier, ce qui crée un point de défaillance unique, les **microservices** utilisent de simples API pour communiquer entre eux
  - ce qui autorise l'intégration avec des applications tierces.

Les Pattern de « Messagerie Asynchrone »

- Objectif : découpler les applications les unes des autres
- On parle de « courtiers de messages » (message broker)
- Le *message broker* **a la responsabilité de livraison** des messages aux destinations appropriées (les consommateurs).

- Avec un *message broker*, l'application source (le producteur) envoie un message à un processus serveur capable d'assurer tout un tas de fonctionnalités en plus de la livraison
  - L'agrégation des données, le routage, la traduction des messages, la journalisation, la persistance, etc.

Protocoles standard ou propriétaires



Message broker : différents modèles de communication

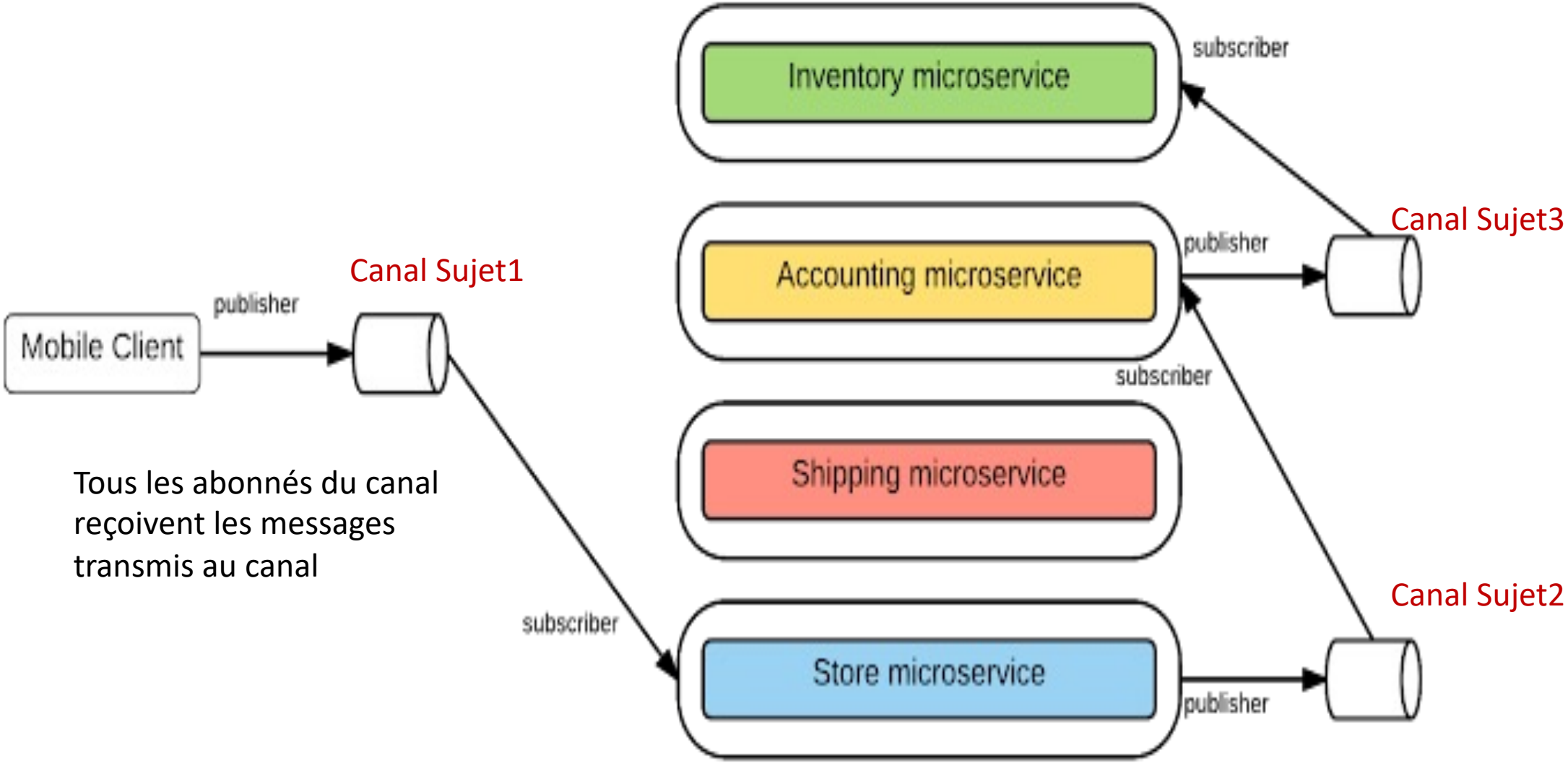
- Publication / Abonnement (sujets)
- Files d'attente
- Many to One
- Requête / réponse asynchrone

Communication par Publication / Abonnement (très utilisé)

Les consommateurs s'abonnent à des **sujets**, et tous les messages publiés sur le sujet sont reçus par tous les abonnés du sujet.

Un abonné peut recevoir le sujet provenant de *n* éditeurs différents. Un éditeur (une application) peut envoyer un sujet à *plusieurs* abonnés.

Ce modèle permet une diffusion simple, axée sur les intérêts, en fonction des sujets auxquels les applications sont abonnées.



Tous les abonnés du canal reçoivent les messages transmis au canal

Les microservices des producteurs **ignorent totalement** les microservices des consommateurs.

Ce style de communication dissocie les *producteurs* des *consommateurs* de messages ; le **courtier de messages** stocke les messages jusqu'à ce que le consommateur soit en mesure de les traiter.

© V. Deslandres, IUT Lyon1

Dans quel cas utiliser le mode Publication / Abonnement ?

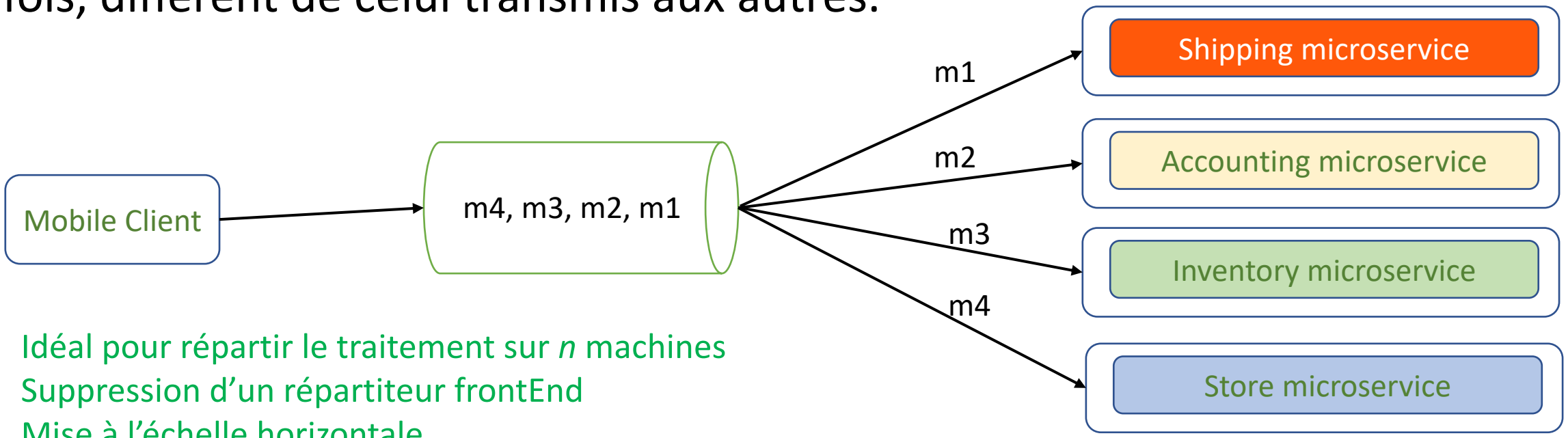
- Quand un **événement engendre plusieurs actions**
  - Ex. soumettre un formulaire : MàJ plusieurs BD, envoyer des notifications, transmettre à des APIs, etc.
- Si on pense devoir **ajouter de nouvelles fonctionnalités dans un futur proche,** sans interrompre ou modifier le code existant
- Ex. achat dans une boutique en ligne
  - Système de commande : enregistre la commande et les données client
  - Système d'inventaire : met de coté l'article, en attente de livraison
  - Système de paiement : tente de débiter le montant de la CB
  - Système de stock met à jour son inventaire

© V. Deslandres, IUT Lyon1

Communication par Files d'attente (très fréquent)

Un producteur de message (une application mobile) envoie des messages à un canal File d'attente FIFO.

Chaque consommateur de message ne reçoit qu'un type de message à la fois, différent de celui transmis aux autres.



- Idéal pour répartir le traitement sur *n* machines
- Suppression d'un répartiteur frontEnd
- Mise à l'échelle horizontale
- Gestion aisée des pics de charge : attente, nombre de consommateurs variables
- Avec une notification de message reçu, on peut remettre le msg non traité dans la file en cas de panne d'un serveur Consommateur

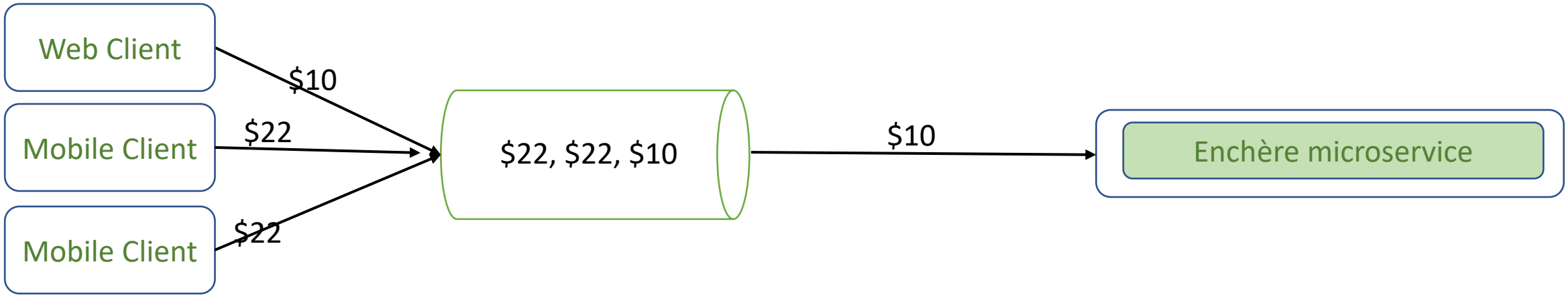
Modèle « machine ouvrière », traitement spécifique à effectuer par des services dédiés

Un cas d'utilisation du mode File d'attente

- Ex.: téléchargement d'images d'un site de partage photos de différentes taille/format
  - Service de traitement d'images : couleur,
  - Service de dimensionnement de l'image
  - Service de conversion en un autre format
  - Service de filtres
- Accusé de réception quand le service a effectué son traitement

Communication Many-to-One (ou « Consommateur exclusif », rare)

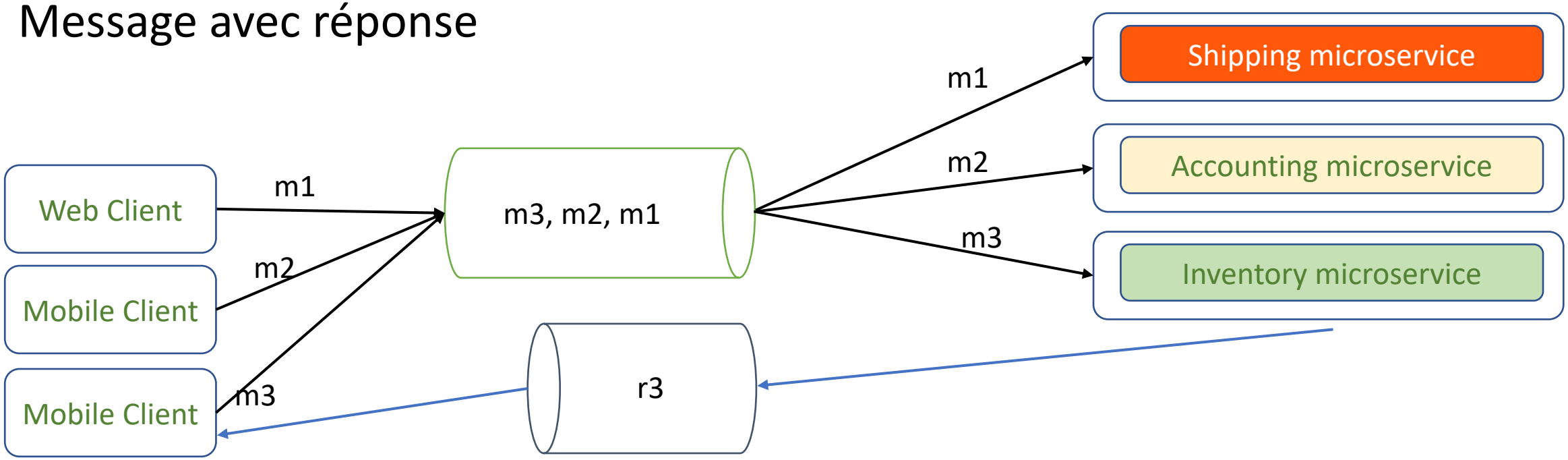
Plusieurs producteurs de messages envoient des messages à un unique consommateur via un canal de type File d'attente FIFO.



- Quand un consommateur doit **sauvegarder l'état du système** entre les messages
- Ou gérer le flux successif des messages dans un **ordre donné**

Communication Requête / réponse asynchrone (peu fréquent)

Message avec réponse



Ressemble à un système synchrone comme en API REST, mais non !
La réponse est jointe au message d'entrée avec le destinataire

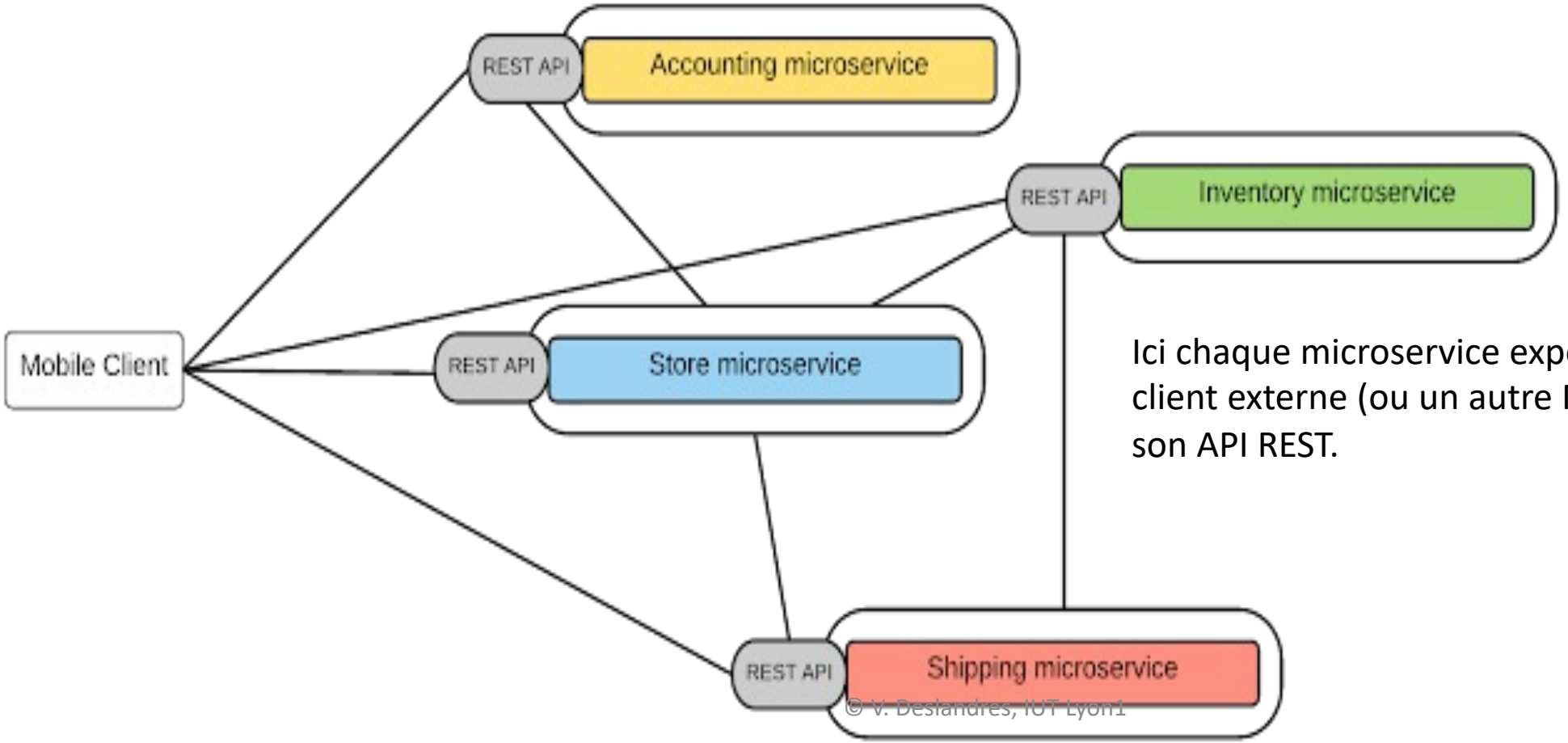
- CAS D'UTILISATION - quand réponse prend du temps (par ex. un utilisateur remplit un formulaire long : écouteur sur le canal de réponse) ou quand on ne sait pas quel serveur va pouvoir traiter le message. Ex. visualiser une vidéo (optimiser, etc.) : processus long

Pattern de communication synchrone

Point à point

Communication point à point (synchrone)

Dans un style point à point, l'intégralité de la logique de routage des messages réside **sur chaque point d'extrémité** : les services peuvent communiquer directement.



Ici chaque microservice expose une API REST et un client externe (ou un autre MS) peut l'invoquer via son API REST.

© V. Deslandres, IUT Lyon1

Principaux inconvénients du style point à point pour la communication de microservices

- Les exigences non fonctionnelles telles que l'authentification de l'utilisateur final, la limitation, la surveillance, etc. doivent être mises en œuvre à chaque niveau de microservice.
- Du fait de la duplication de fonctionnalités communes, chaque implémentation de microservices peut devenir complexe.
  - Peu adapté aux implémentations de microservices à grande échelle
- Il n'y a aucun contrôle sur la communication entre les services et les clients (même pour la surveillance, le traçage ou le filtrage)

Une solution ?

- Disposer d’un **bus de messagerie central léger** qui peut fournir une couche d'abstraction pour les microservices
- Qui peut être utilisé pour implémenter diverses contraintes non fonctionnelles.

- Ce style de communication est **l’API Gateway.**

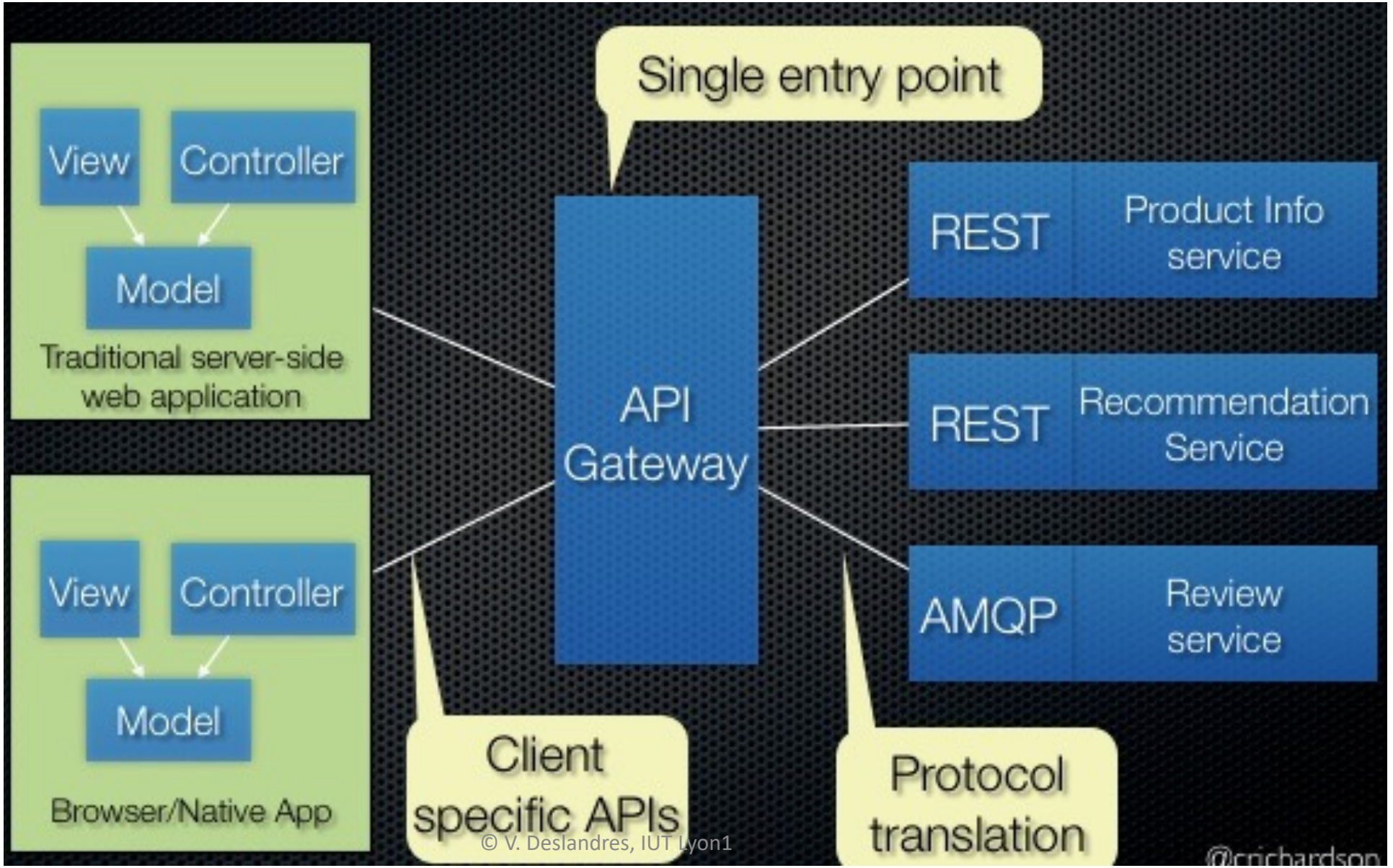
Pattern Passerelle d'APIs

API Gateway

Pattern API Gateway

- IDEE : utiliser une passerelle comme point d'entrée principal pour tous les clients / consommateurs avec les exigences non fonctionnelles communes.
- En général : gérée via REST / HTTP.
- Les fonctionnalités des microservices sont exposées via l'API-GateWay, et sont des API gérées par la passerelle.

Passerelle API



L’API GW : avantages

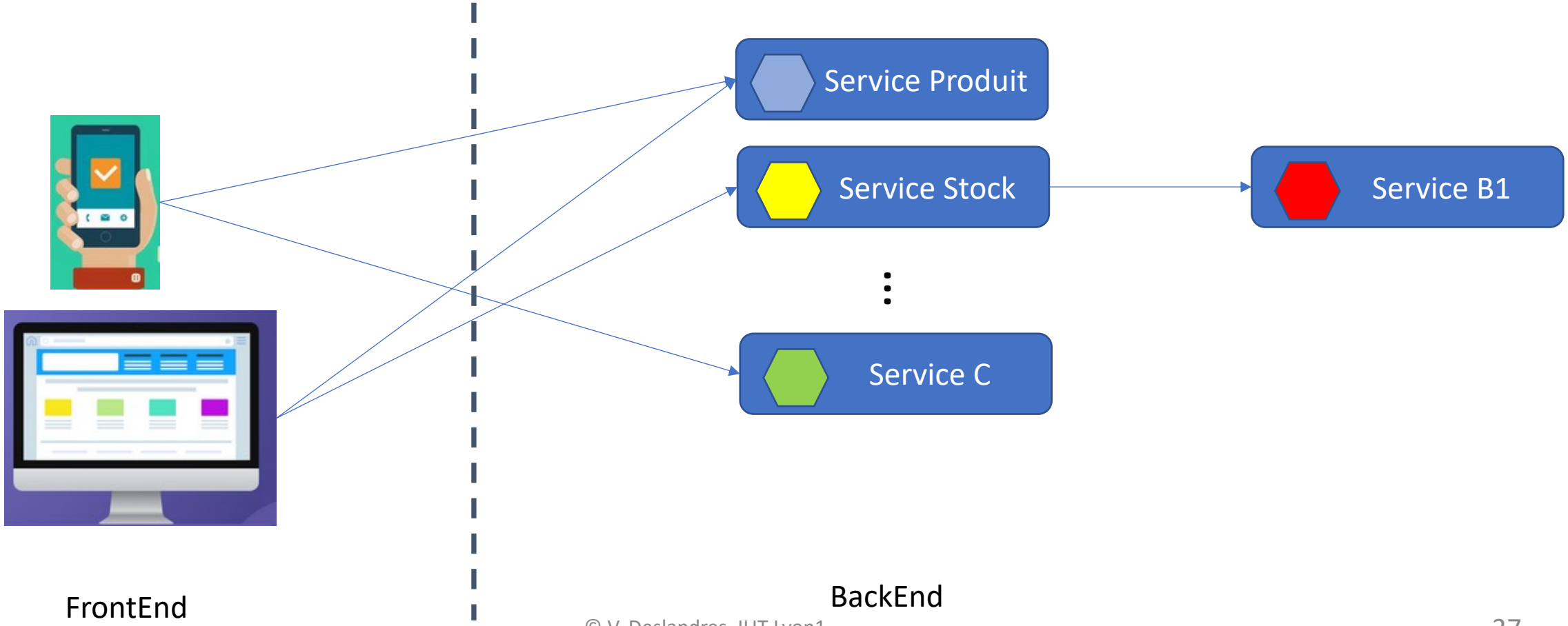
- Découplage App Web Client (Front) et microservices (Back)
  - C’est la passerelle qui permet l’accès aux services
- Fournir les abstractions requises au niveau de la passerelle pour les microservices existants. Par exemple, plutôt que de fournir une API de style unique, la passerelle API peut exposer une API différente pour chaque client.
- Routage / transformations de messages légers au niveau de la passerelle
  - Equilibrage de charges défini au niveau de la passerelle, filtrage des requêtes (par ex., liste blanche d’adresses IP pouvant accéder aux MS), entêtes ou agrégation des résultats

L’API GW : avantages (2)

- C’est l’endroit unique où l’on peut définir des contraintes non fonctionnelles telles que la **sécurité, la surveillance et la limitation**
  - Ex. logique d’authentification centralisée, limitation du taux d’usage de chaque MS (ex. moins de 100 appels à la seconde)
- Avec l'utilisation du modèle API-GW, le microservice est encore plus léger car un grand nombre d’exigences sont implémentées au niveau de la passerelle
  - Ex. mise en cache des réponses, relance des requêtes en cas d’échec, mise en place du pattern Court-Circuit (moins de code dans les MS), journalisation de l’utilisation du MS

Illustration Passerelle API (API Gateway)

Comment faire correspondre les services du FrontEnd avec ceux du BackEnd en toute sécurité et rapidement ?



© V. Deslandres, IUT Lyon1

Illustration API Gateway

- Exposer les adresse IP…?
  - Si adresse IP et numéro de Port exposés publiquement pour chaque service de BackEnd, porte ouverte aux hackers : **problème de sécurité**
  - 2e problème : la **latence**. Si une page a besoin des informations de 5 MS, le FrontEnd a besoin de 5 connexions aux services, le temps de réponse peut être long.

- Le Proxy Inverse (*reverse proxy,* protection du serveur) résout les 2 problèmes précédents
  - Les adresses IP des services sont privées; seule l'adresse IP de la passerelle API est exposée. Une authentification sera nécessaire pour accéder aux MS au niveau de la passerelle API (code non dupliqué au niveau des services).
  - Résout le problème de latence : la passerelle joue le rôle d'Agrégateur, avec des connexions simultanées, ce qui réduit la durée de réponse.

© V. Deslandres, IUT Lyon1

Point faible de l'API Gateway – Les Outils

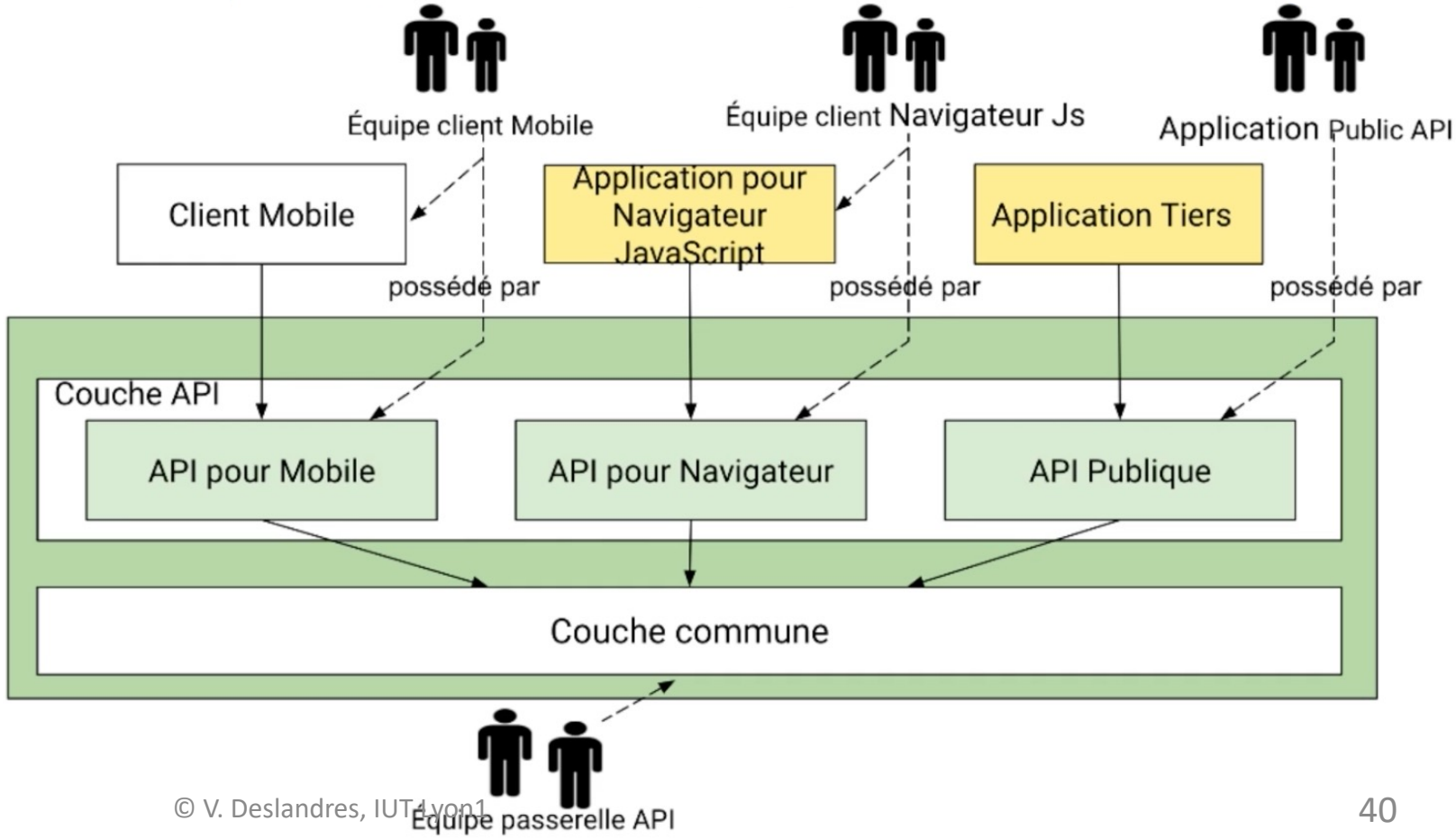
- Fragilité de l'architecture : unique point de défaillance de l'application
- Solution : une **mise à l'échelle horizontale**
  - On dédouble les instances de la passerelle et on ajoute un équilibreur de charge qui gère le fonctionnement de ces instances

- Cette solution est la plupart du temps mise en œuvre dans les **plateformes de gestion et d'administration des API**, proposées par les fournisseurs de services Cloud
  - Apigee, MuleSoft, Axway, Akana API Management, Young App, Oracle API Gateway, TIBCO Exchange Gateway, etc.
  - Open source : Tyk.io, Kong Gatway, Ocelot

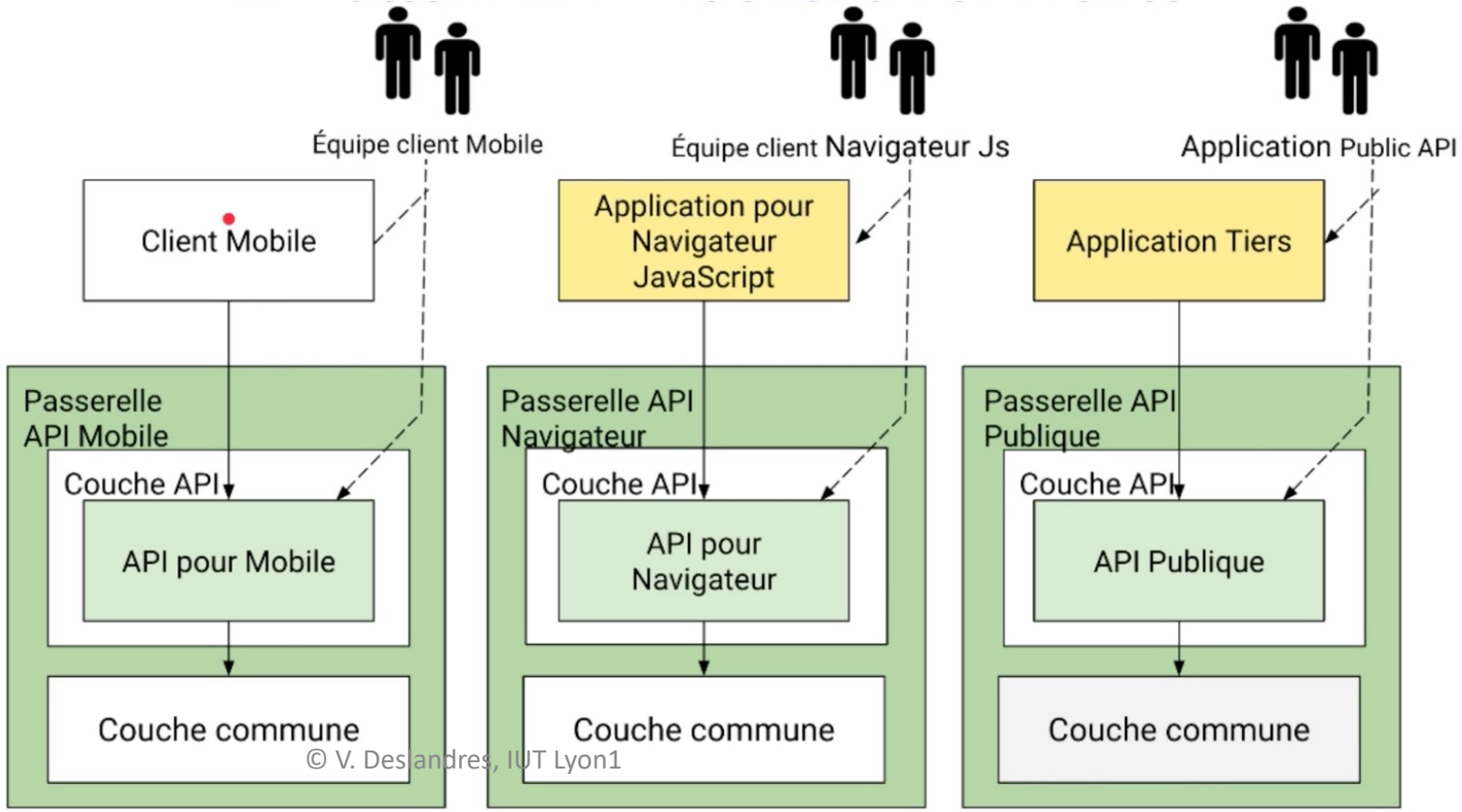
Pattern « Backend for FrontEnd » (BFF)

- Amélioration de l'API Gateway
- Constat : une passerelle API a une responsabilité floue

Ex.: on peut avoir **plusieurs équipes** qui travaillent pour une utilisation donnée de l'API GW



- Les équipes doivent se coordonner tout au long du processus de développement
  - Par ex., s'accorder pour la livraison d'un changement
  - Cette situation rappelle l'application monolithique
- Le **pattern BFF** consiste à développer une API par type de Client



API indépendantes mais **même socle technologique**

Une bibliothèque partagée pour la couche commune

© V. Desjardres, IUT Lyon1

Avantages du Pattern BFF

- **Fiabilité** : les APIs sont isolées les unes des autres. Quand l'une fonctionne mal, pas d'impact sur les autres.
- **Observabilité** : chacune tourne dans un processus indépendant. Plus facile de voir la responsabilité de chacune.
- **Evolutivité** : plus simple
- **Temps de démarrage plus court :** taille réduite des API-GW

© V. Deslandres, IUT Lyon1

Quizz !

1. La composition par agrégation est sans doute la forme de composition des microservices la plus répandue et la plus utilisée dans les applications à base de microservices.

   L'aggrégateur se trouve du côté de l'application FrontEnd ou du BackEnd ?

2. Une passerelle API peut prendre en charge plusieurs fonctionnalités au sein d'une application à base de de microservices, mais à la base, elle est mis en place pour résoudre 2 problèmes. Lesquels ?

Quizz (suite)

3. Vous êtes responsable du développement d'une application de commerce électronique avec une architecture microservices. Vous avez la charge de mettre en place un système de communication entre le service Commandes et le service Client.

   A chaque fois que le service Client met à jour les informations d'une personne, le service Commande met aussi à jour les données du client dans sa base de données. Les 2 services ont des bases de données séparées.

   Quel modèle de messagerie est adapté au à ce scénario ?

Pattern Idempotent Consumer

Service idempotent

Pattern « Idempotent Consumer »

Concept identique à *l'idempotence* des fonctions mathématiques (ex. valeur absolue)

Contexte

- Dans une application, il est fréquent d'utiliser un courtier de messages (*message broker*) qui garantit au minimum que le message sera livré à un service client même si une erreur apparait.
- Effet de bord : le client peut être invoqué de façon répétée pour le même message, et les informations transmises seront alors différentes (si fonction du temps) ou cela peut générer des bugs
  - Un client d'un message DébiterCompte(montant) risque d'être débité plusieurs fois…

- On appelle ***Idempotent*** un service Client pour lequel le fait de recevoir plusieurs fois le même message ou une seule fois, produit le même résultat.

Comment gérer la duplication de messages ?

- Certains clients sont naturellement idempotents
  - Exemple : ceux qui contrôlent, stockent, tracent les messages
- Pour les autres, il faut filtrer les messages reçus et détecter ceux qui sont des duplications de messages déjà reçus et traités.
- Une solution est de sauvegarder en BD les IDs des messages qui ont été traités.
  - Quand arrive un nouveau message, on va voir en BD s'il a déjà été traité.
- Où stocker ces IDs ?
  - Soit utiliser une table séparée des PROCESSED_MESSAGES
  - Soit on stocke les IDs dans les objets Métiers (Business entities) que le message crée ou met à jour.

Des exemples de code de Chris RICHARDSON ici: http://eventuate.io/exampleapps.html

© V. Deslandres, IUT Lyon1

Pattern Schema Registry

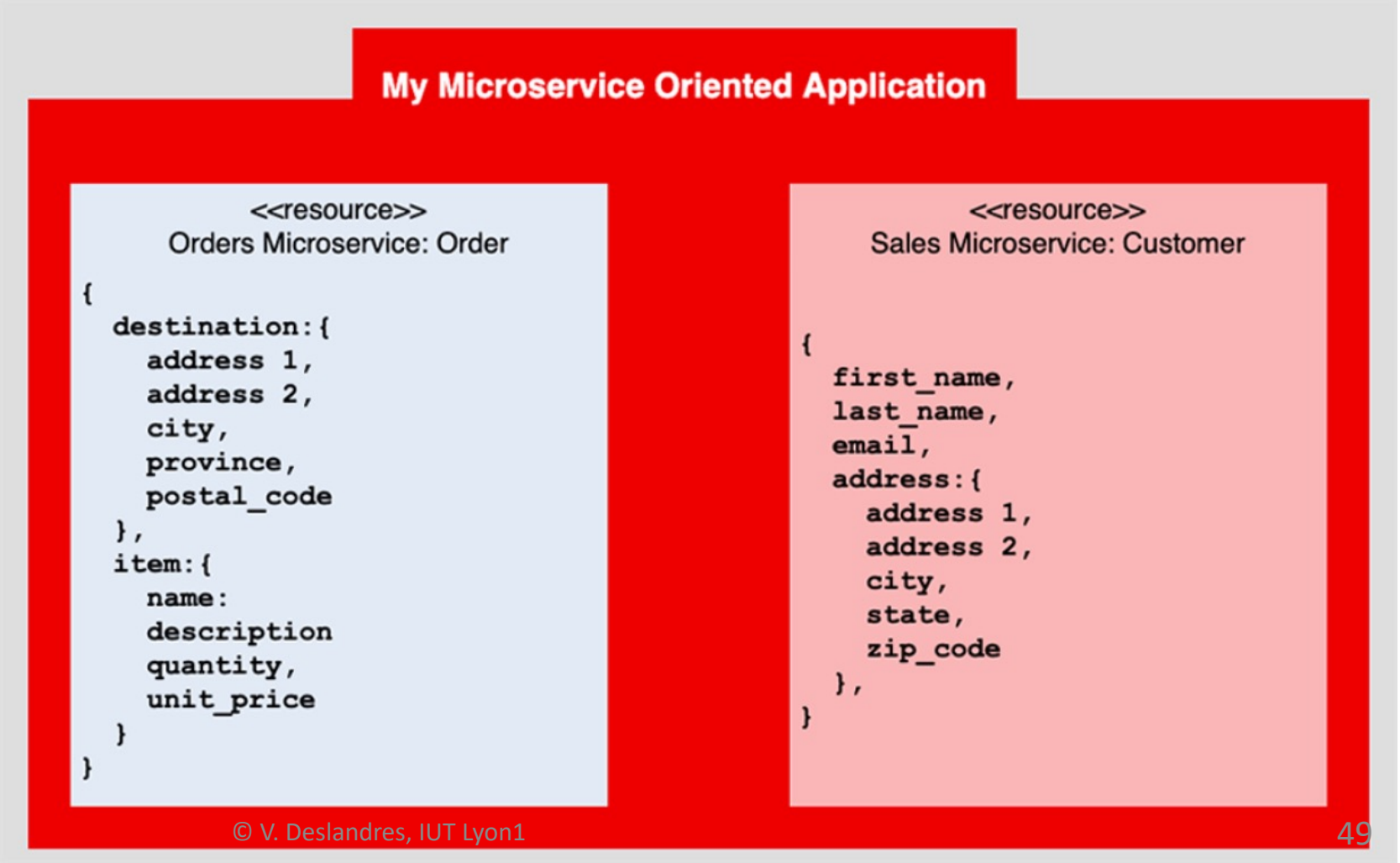
Schéma de définition des données

Problème de la consistance des données en architecture MS

Ex.: une commande et son client

Les 2 schémas de données n'ont **pas les mêmes noms** (Order.*destination* et Customer.*address, postal/zip_code, etc.*) pour la localisation

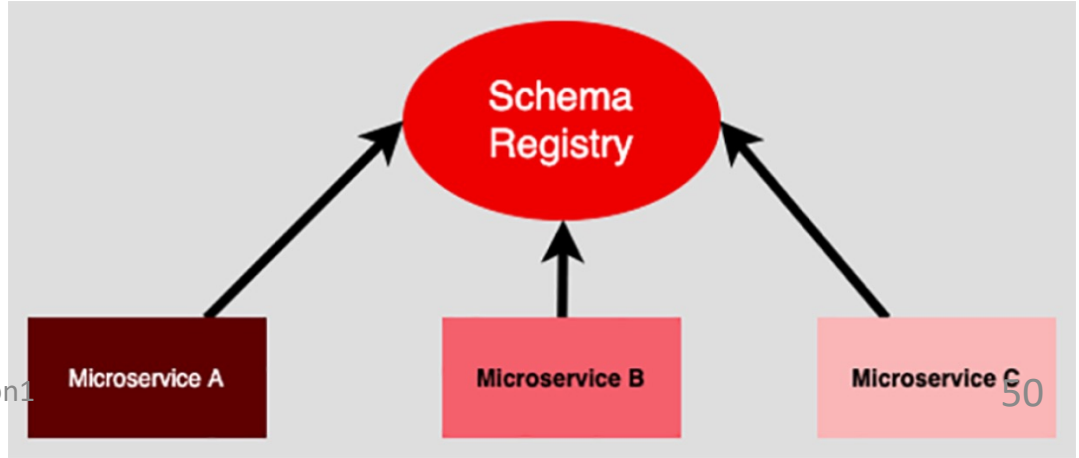
Mais **même contenu**.



- L'humain est capable de déceler la similarité sémantique des données, mais la machine s'arrête à la syntaxe.
- Il existe des *outils* comme BizTalk Server (inclus dans Visual Studio de MicroSoft) qui permettent de définir les équivalences. Mais il faut un humain pour implémenter ces translations
  - Ex.: mapper **postal_code** en **zip_code**

- **Une autre solution est d'utiliser un Schema Registry.**
- **Un Schema Registry** centralise la description des données pour de multiples micro services. C'est l'unique *source de vérité* en termes de définition du schéma des données.

Permet de connaitre le schéma des données avant d'encoder les échanges : gain de temps



Autre cas d'utilisation du Schema Registry

- Pour l'étape de Validation de code
  - Eviter le problème des tests qui ne fonctionnent plus parce qu'un DBA a décidé de modifier la structure des données…
- On crée un **code de validation** qui permet aux microservices d'être toujours alignés avec la dernière version de définition des données du domaine

```
validateData(data, schemaRegistryUrl.schemaName) {

  // récupérer le schéma du Registry

  // comparer la data au schéma récupéré

  // si la data ne correspond pas à la définition du schéma, lever une erreur
}
```

© V. Deslandres, IUT Lyon1

Observation de Martin Fowler

- Les Patterns ont toujours 2 parties : le COMMENT et le QUAND.
- On ne doit pas simplement savoir comment les implementer, il faut aussi savoir QUAND les utiliser et QUAND les ignorer.

© V. Deslandres, IUT Lyon1

Le pattern MV-VM

MVC avec un Modèle de Vue en plus

Architecture MV-VM

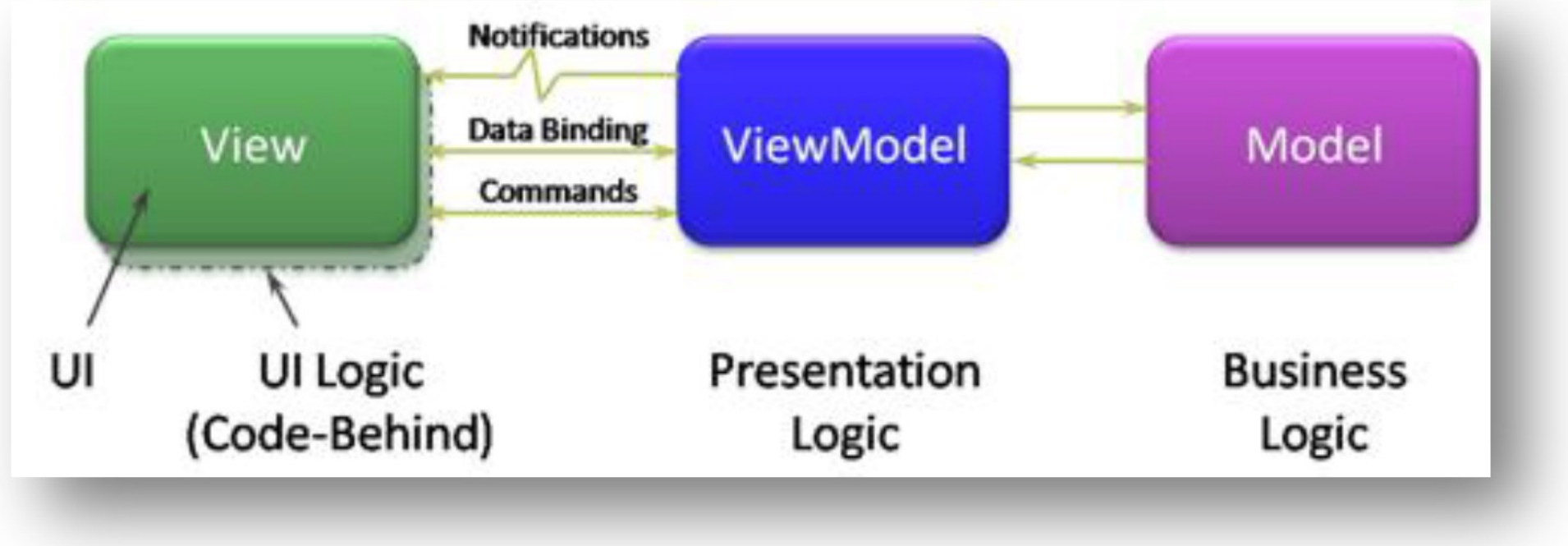
- Objectif : découplage fort entre le code et l'IHM dans une apps
- Prend en compte la notion de **Service** et d'**Injection de Dépendances**
- Initialement proposé par un FWK, .NET en 2005, puis agrégation des méthodes de découpages sémantiques ou fonctionnels du code issus de différentes approches et le terme générique MV-VM a été conservé
  - Notion beaucoup plus vaste aujourd'hui, MV-VM concerne le découplage fort
  - Techniques de communication "sans contact" entre les codes

M-V-VM

- **Modèle** : les données et la logique métier. Complètement indépendant de l'IHM
- La **Vue** : un ensemble d'éléments visuels. C'est l'IHM, décrite en Xaml. Gestion de tâches de la Vue : drag'n drop, zoom, etc.
  - La vue peut contenir un peu de code-behind notamment pour gérer la logique propre à l'interface
- le **ViewModel** (*Model of a View,* Modèle de Vue) : intermédiaire
  - Code qui permet de **bien présenter** les données
  - Certains traitements sur les données ne peuvent pas être placés dans la Vue, ni dans les Données (limites des procédures stockées)
  - Le VM va stocker les états de l'IHM et faire des traitements sur les données en entrée ou en sortie ; code la « logique de présentation »
  - C'est une **abstraction de la Vue** ou une **une spécialisation du Modèle** que la Vue peut utiliser (data binding)

Notion de commande du pattern

- **Commande** = une action que l'utilisateur peut réaliser depuis l'interface
  - Clics sur des objets, saisie de texte, les mouvements de doigts sur un écran tactile, etc.



Exemple en C#

- Imaginons une application qui a besoin d'utiliser des nombres aléatoires.
- Le premier ViewModel concerné va très certainement contenir un code du type :

```
public class MainViewModel : INotifyPropertyChanged {
        public event PropertyChangedEventHandler PropertyChanged;
                                                                    (modèle WPF)
        private Random random = new Random();
        private void UneMéthode() {
                var i = random.Next(0,5);
                // blabla
        }
        // ...
}
```

Observations de ce code

- Si l'application contient 10 ou 20 ViewModel ou autres codes (dans les Models par exemple) faisant appel à des nombres aléatoires, cela va poser plusieurs problèmes : lesquels ?
  1. La création de plusieurs instances de Random
  2. La multiplication d'un code qu'on croyait local et unique
  3. Couplage fort vers Random : si demain on souhaite changer de générateur aléatoire : ??

➔ Séparer ce code en 2

- La génération de nombre aléatoire
- Les classes qui l'utilisent

```
public class MainViewModel : INotifyPropertyChanged
{
        public event PropertyChangedEventHandler PropertyChanged;

        private void UneMéthode() {
                var cr = new ClassicRandom();
                var i = cr.GetNext(0,5);
                // blabla
        }
        // ...
}
```

```
public class ClassicRandom
{
        private Random random =new Random();

        public int GetNext(int main, int max) {
                return random.Next(min, max);
        }

}
```

```
public class SecondViewModel : INotifyPropertyChanged
{
        public event PropertyChangedEventHandler PropertyChanged;

        private void UneAutreMéthode() {
                var cr = new ClassicRandom();
                var i = cr.GetNext(12, 57);
                // blabla
        }
        // ...
}
```

A quel(s) pb(s) a-t-on répondu ici ?

Pour répondre au pb (1) instances multiples de Random

- Créer une classe **Static** ClassicRandom ?
  - Non ça n'est pas *thread-safe*

- **Ajouter un constructeur** à la classe ClassicRandom, avec un paramètre du genre entier pour choisir la méthode de génération aléatoire ?
  - *Avec* un Switch ou des If/Else dans le futur …
  - *Non, le code spaghetti s'installe, les If/Else vont se succéder, les Switch … jusqu'au moment où on devra porter le code sous une autre plateforme et qu'on s'apercevra que le cas #12 ne passe pas la compilation sous Xamarin…*

Une idée ?

Indice : il n'y a pas de code fortement découplé sans utilisation des interfaces

Découpler avec une interface…

- Une interface : permet des implémentations différentes de Random sans if/else, ni Switch
- « La programmation objet devrait être une programmation sans IF »

```
// fichier IRandomProvider.cs

public interface IRandomProvider
{
    int GetNext(int min, int max);
}
```

```
// une classe d'implémentation :
// fichier ClassicRandom.cs

public class ClassicRandom : IRandomProvider
{

    private Random random = new Random();

    public int GetNext(int min, int max) {
        return random.Next(min, max);
    }
}
```

… et une Factory

C'est la Factory (un Singleton) qui va instancier le générateur qui nous convient :

```
// fichier RandomFactory.cs

public class RandomFactory {
   private RandomFactory() {}                // constructeur privé
   private static RandomFactory instance; // une seule instance en static
   private IRandomProvider provider;         // c'est le générateur choisi

   public static RandomFactory Instance {
       get { return instance ?? (instance = new RandomFactory()); }
   }
   public IRandomProvider GetProvider => provider ?? (provider = new
ClassicRandom());
}
```

Ce qui donne pour les classes ‘clientes ’

```
// fichier MainViewModel.cs

public class MainViewModel : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;

private void UneMéthode() {
        var i = RandomFactory.Instance.GetProvider.GetNext(0,5);
        // blabla
    }
    // ...
}
```

```
// fichier SecondViewModel.cs

public class SecondViewModel : INotifyPropertyChanged {
        public event PropertyChangedEventHandler PropertyChanged;

        private void UneAutreMéthode() {
                var i = RandomFactory.Instance.GetProvider.GetNext(12, 57);
                // blabla
        }
        // ...
}
```

On passe d’un couplage fort vers *ClassicRandom* à un couplage faible (vers l’interface au sein de la Factory)

A-t-on répondu aux 2 autres pbs ?

Réponse aux problèmes : (1) instances multiples de Random et (3) facilité de changement de générateur

(1) par le Singleton et le provider

(3) Changement ici, pas chez les classes Clientes

```
// fichier RandomFactory.cs

public class RandomFactory {
   private RandomFactory() {}
   private static RandomFactory instance;
   private IRandomProvider provider;

   public static RandomFactory Instance {
       get { return instance ?? (instance = new RandomFactory()); }
   }
   public IRandomProvider GetProvider => provider ?? (provider = new
ClassicRandom());
}
```

Par ex.: provider = new CryptoRandom()

© V. Deslandres, IUT Lyon1

Pour un découplage encore plus fort

- Si on a plusieurs services à gérer, pour éviter d'avoir *n* factories différentes pour les *n* services (messages à afficher, service de mail, générateur aléatoire, etc.)
  - On fait appel à l'**injection de code (IoC)**
- On remplace la *Factory* par un conteneur, par exemple un *repository* ou un *Dictionnaire<key,value>* avec une clef **par service requis** *ou un fichier XML qui associe le nom physique de chaque service à une clef*
  - Tous les services (dont *ClassicRandom* et *CryptoRandom)* sont enregistrés dans le conteneur IoC.
  - *MainViewModel* et *SecondViewModel* spécifient le service à utiliser dans leur constructeur

Les classes Lazy<> du C#

- En utilisant Lazy<IService> au lieu de IService directement, on va laisser l'injonction de dépendance faire son job en évitant toutes les créations d'instances en cascade qui peuvent en découler.
- Les instances des services ne sont créées que lorsqu'on s'en sert
  - Inutile s'ils sont utilisés tout au long de l'application !

```
public class MainViewModel {
        private Lazy<ISomeRepository> _lazySomeRepository;
        private ISomeRepository SomeRepository =>
_lazySomeRepository.Value;
        // Constructeur
        public MainViewModel(Lazy<ISomeRepository> lazySomeRepository){
                                        _lazySomeRepository =
lazySomeRepository; }
        public void RefreshData(){
           var allData = SomeRepository.GetAllData();
           .....
        }
}
```

Source : blog d'Olivier DAHAN https://www.e-naxos.com/Blog/post/Ode-a-la-paresse.aspx