

07/10/2023

Version 2

# SPRING CLOUD

JÉRÉMY PERROUULT

A decorative wavy line in light blue and white, flowing vertically along the left edge of the slide.

# INTRODUCTION

INTRODUCTION

# INTRODUCTION

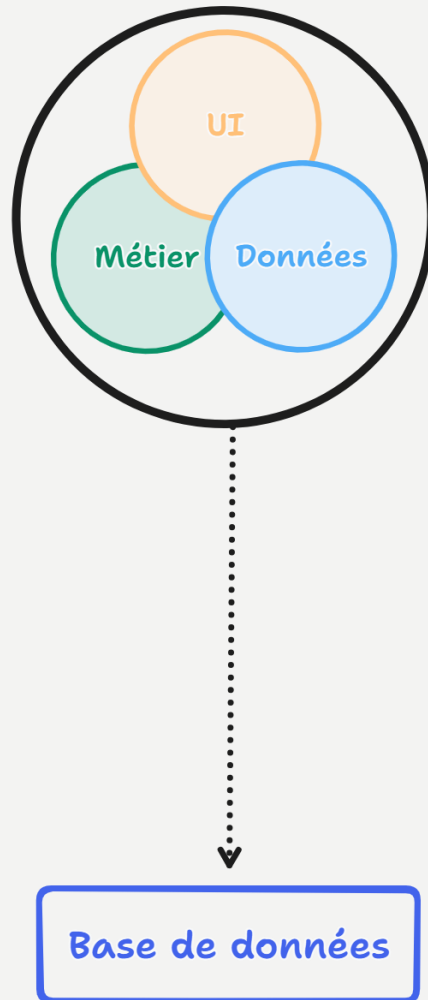
- Les **microservices**
  - Architecture(s)
  - Approche de développement
- Décompose l'application en éléments plus simples et indépendants
  - Certains services ont besoin d'autres services : besoin de communication entre eux
- On parlera des **microservices backend**
  - Pour le front, on parle de **micro-frontend**, sujet qui ne sera pas abordé ici

# INTRODUCTION

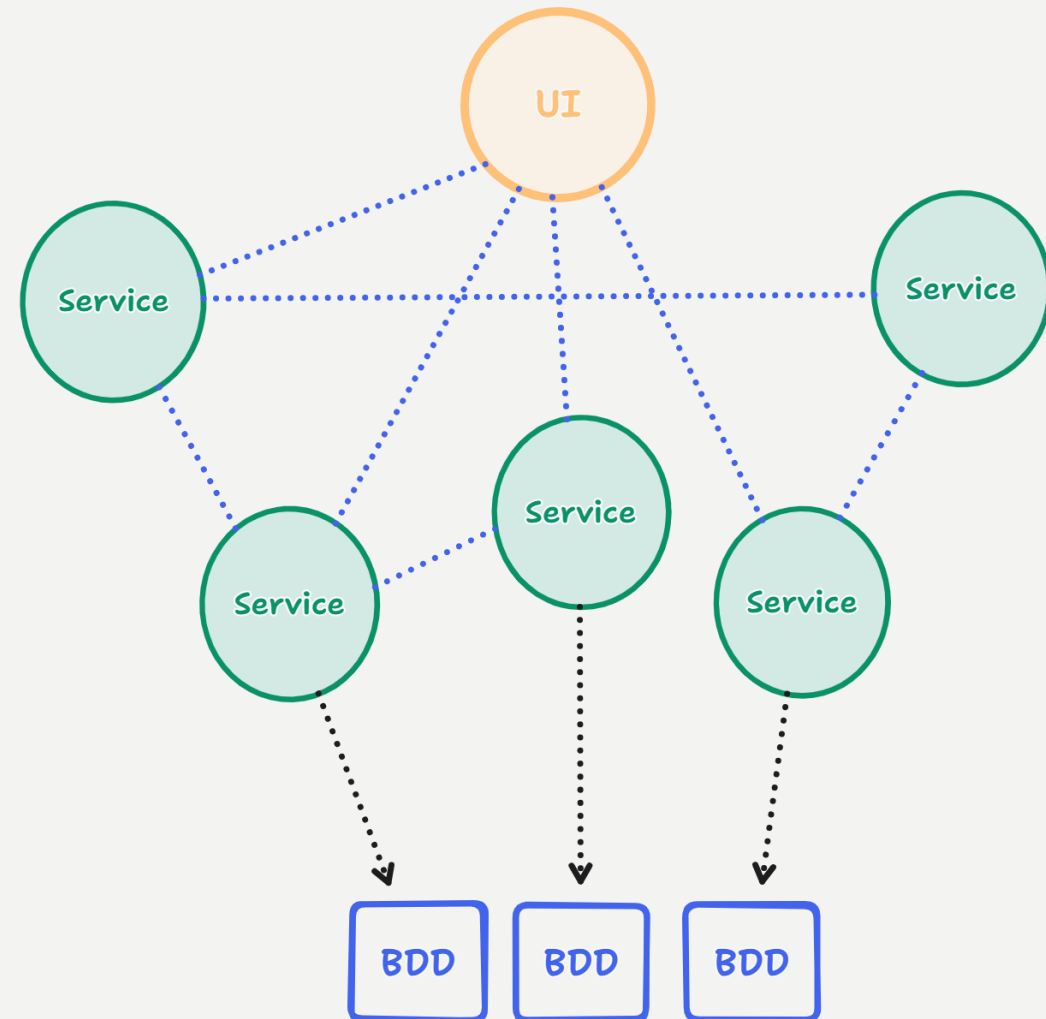
*NOTE : Ceci est un exemple.*

*Il existe une multitude d'architectures en microservices*

Architecture Monolithique



Architecture Microservices



# INTRODUCTION

- Fonctionnent principalement par **API REST**
  - Chaque service expose des fonctionnalités
  - Chaque service peut consommer une fonctionnalité exposée par un autre service
  - Chaque service peut agréger des informations d'autres services
  - Etc.
- En gardant le même vocabulaire **HTTP REST**
  - GET            Récupérer des informations
  - POST          Ajouter des informations
  - PUT            Modifier des informations
  - PATCH        Modifier partiellement des informations
  - DELETE       Supprimer des informations

# INTRODUCTION

- Les architectures en **microservices** sont généralement plus complexes et plus demandeuses en ressources
  - Patterns
    - Pour nous aider à faire communiquer les **microservices** entre eux
  - Technologies
    - Ensemble de **librairies** et **frameworks** pour nous aider à résoudre les problèmes les plus communs (souvent des implémentations des Patterns)

# INTRODUCTION

- Mais présente un réel avantage
  - Séparation des briques fonctionnelles (même si, en développement plus classique, le découpage en modules est possible)
  - Séparation des équipes
  - Evolutivité / maintenabilité
  - Adaptation des ressources
  - Gestion des erreurs

A decorative wavy line in light blue and white, flowing vertically along the left edge of the slide.

# ARCHITECTURE

ARCHITECTURE IDÉALE



# ARCHITECTURE

- En **microservices**, on peut implémenter différents styles d'architectures
  - **MAIS**, et malgré des dépendances inévitables entre services
  - On doit tendre vers des services indépendants les uns des autres (couplage faible)
    - Pour limiter les effets de bord
      - En cas d'évolution, ou de bugs
    - Pour mieux gérer le développement
    - Pour mieux gérer le déploiement
    - Pour mieux gérer la scalabilité
    - Pour mieux gérer les erreurs
  - On doit avoir des services résilients, capables de maintenir leur bon fonctionnement indépendamment d'autres services qui seraient eux, plus disponibles

# ARCHITECTURE

- Ce qui veut dire ... qu'on refera une classe, pour designer un même domaine, dans différents services ... (exemple **JAVA**)

```
@Getter @Setter
@ToString
public class Produit {
    private int id;
    private String nom;
    private Double prix;
    private int note;
}
```

# ARCHITECTURE

- ... en langages différents, ou identiques (exemple **C#**)

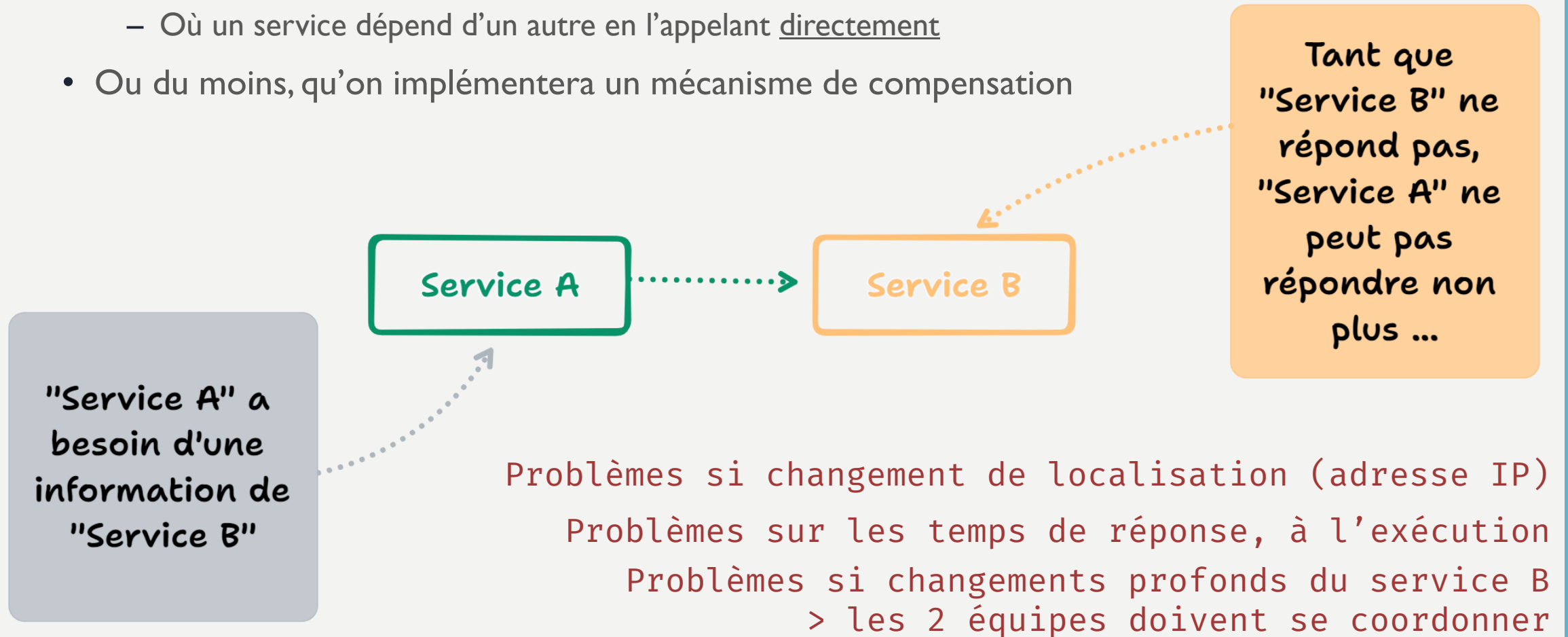
```
public class Produit
{
    public int Id { get; set; }
    public string? Nom { get; set; }
    public double? Prix { get; set; }
    public int Note { get; set; }
}
```

# ARCHITECTURE

- Et le principe **DRY** ? Tenté d'utiliser une bibliothèque partagée ?
  - Vous vous retrouverez avec un couplage fort
    - Une modification, (ex : ajout ou retrait d'une nouvelle information) aura des répercussions dans tous les services
    - Et occasionnera éventuellement des défaillances

# ARCHITECTURE

- Ce qui veut dire aussi ... qu'on évitera ce genre de situation ...
  - Où un service dépend d'un autre en l'appelant directement
- Ou du moins, qu'on implémentera un mécanisme de compensation

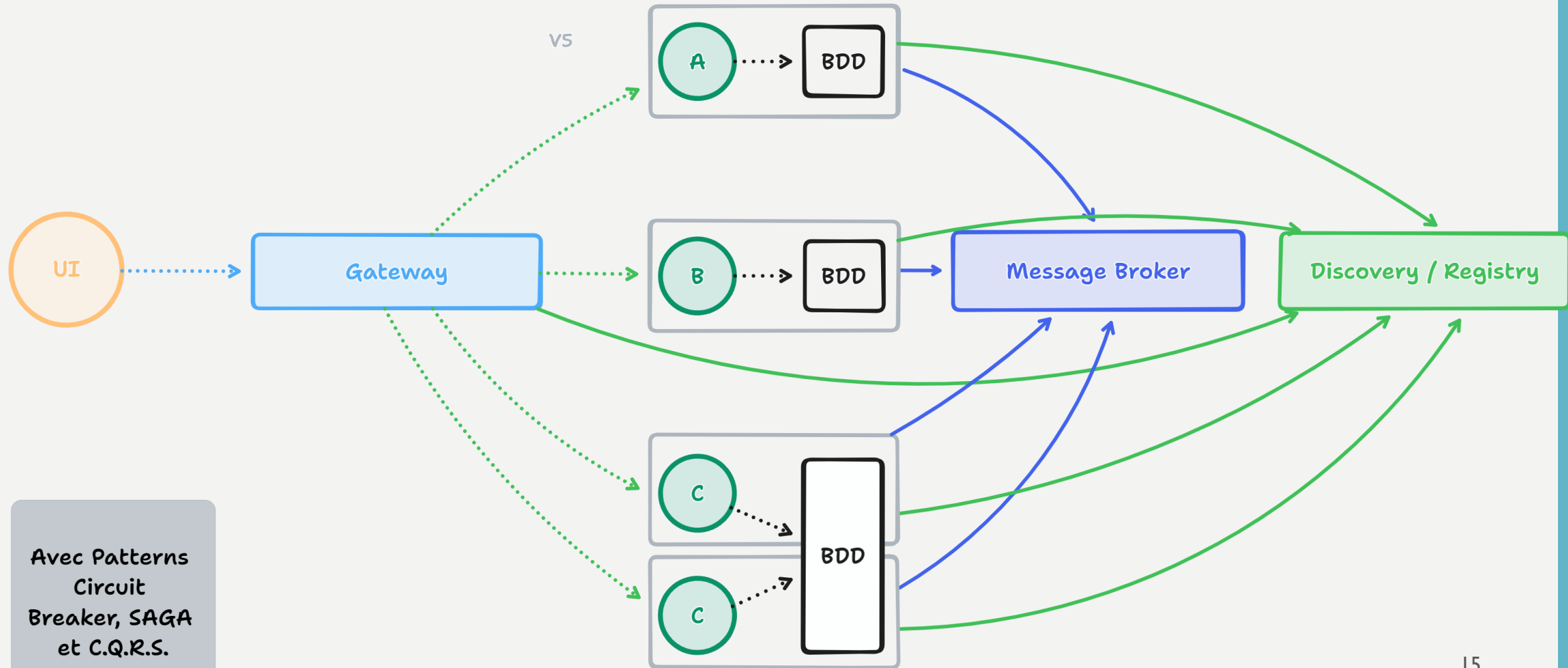


# ARCHITECTURE

- Une base de données, ou plusieurs bases de données ?
  - Un partage des ressources peut limiter les performances d'un ou des services
  - Une modification dans la structure des données peut impacter les services qui en consomme une partie (idem classes partagées)
- DONC, on privilégiera d'utiliser une base de données par service
  - En ayant conscience de perdre l'avantage des transactions et des contraintes référentielles
    - Qu'il faudra compenser par d'autres mécanismes

# ARCHITECTURE

- Pour finalement tendre vers cette architecture





# APPLICATION

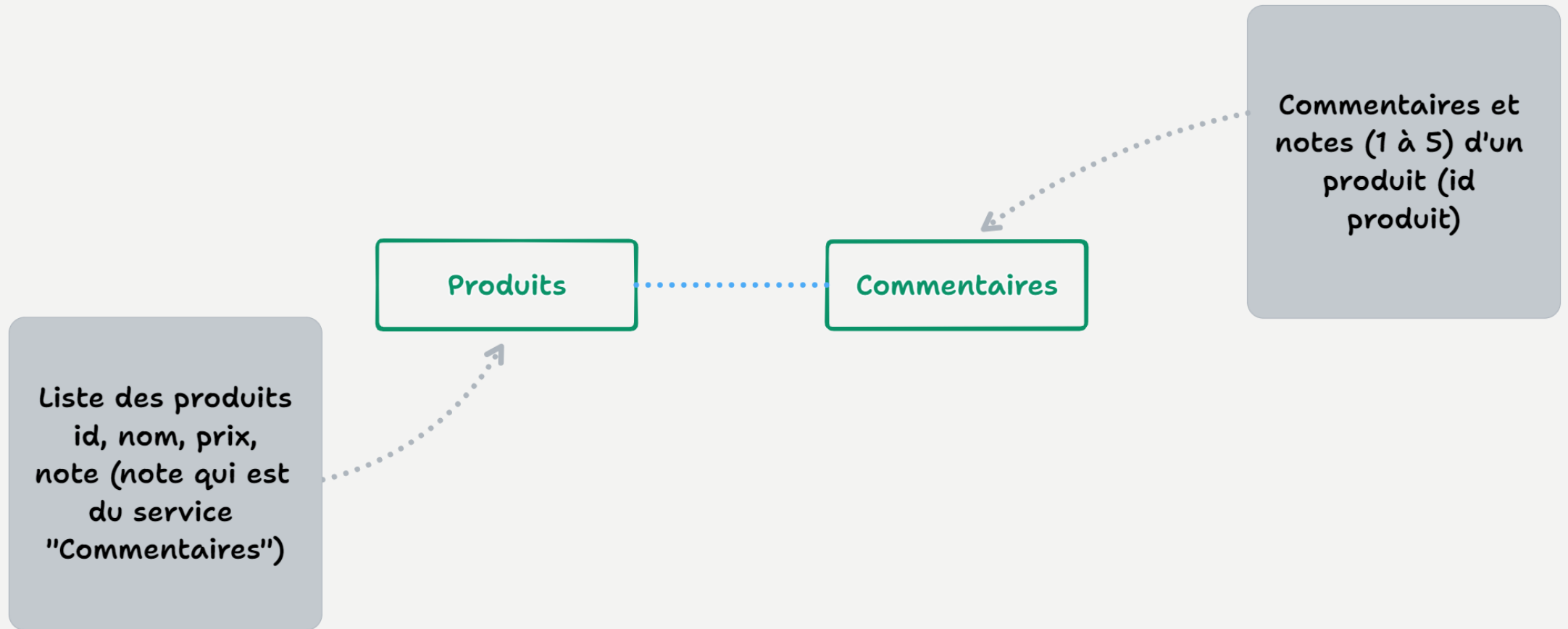
DEUX PREMIERS SERVICES



# APPLICATION

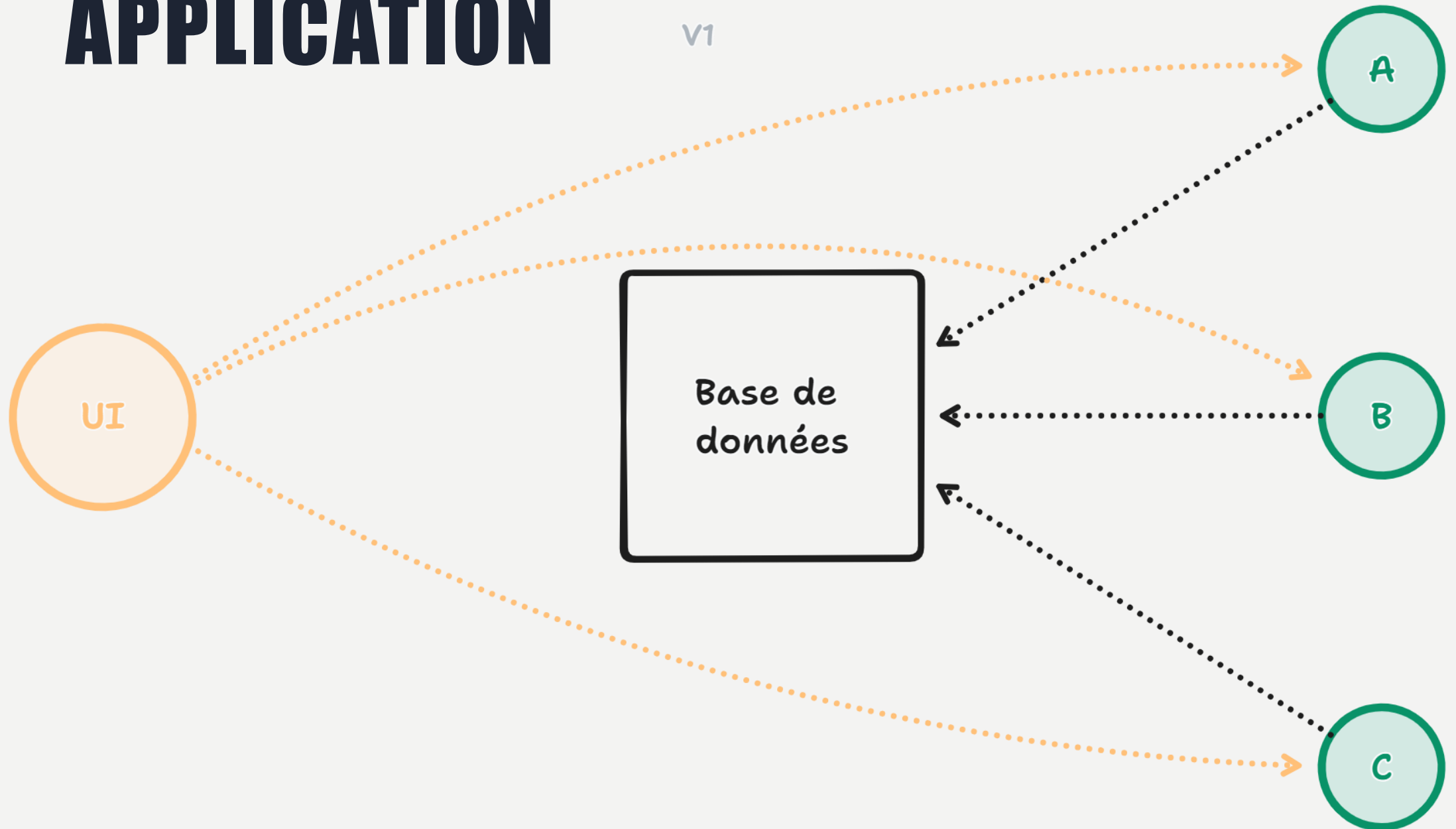
- Fabrication d'une application e-commerce
  - Des produits
  - Des commentaires
  - Des clients
  - Des commandes
  - Des paiements
  - Des expéditions

# APPLICATION



# APPLICATION

v1



# APPLICATION

## Modèle de données

- > **Produit** id, nom, prix, notable (oui / non)
- > **Commentaire** id, texte, note (0 à 5), produit (id du produit)

## Règles de gestion

- > Un commentaire ne peut exister QUE si le produit est « notable »
- > Un produit ne peut pas être supprimé si des commentaires existent pour lui

# APPLICATION

## Service **Produit**

- > Récupérer la liste des produits, avec la note moyenne de chacun produit
- > Ajouter un produit

## Service **Commentaire**

- > Récupérer un commentaire, avec le nom du produit
- > Ajouter un commentaire pour un produit

# APPLICATION

- Mettre en place l'architecture précédente
- Utiliser une base de données **PostgreSQL**, ou **MySQL**, ou **HSQL** ...