

# Introduction aux architectures micro services

Cours Architectures distribuées

Alexandre Touret

© Creative Commons Attribution 4.0 International

# Alexandre Touret

Architecte logiciel à Worldline

 <https://blog.touret.info>

 <https://github.com/alexandre-touret/>

 <https://www.linkedin.com/in/atouret/>

# Introduction

Ce cours vise à introduire les concepts fondamentaux des architectures microservices.

Les points abordés seront:

- Fondamentaux (Pourquoi? Les défis, comment les concevoir)
- Protocoles (HTTP, REST, GRAPHQL, GRPC)
- Couplage lâche
- Utilisation de messageries asynchrones
- Quelques patterns : API Gateway, Back For Front
- Transactions distributées et cohérence des données
- Monitoring & Observabilité (Optionnel)
- Sécurité (Optionnel)

# Fondamentaux

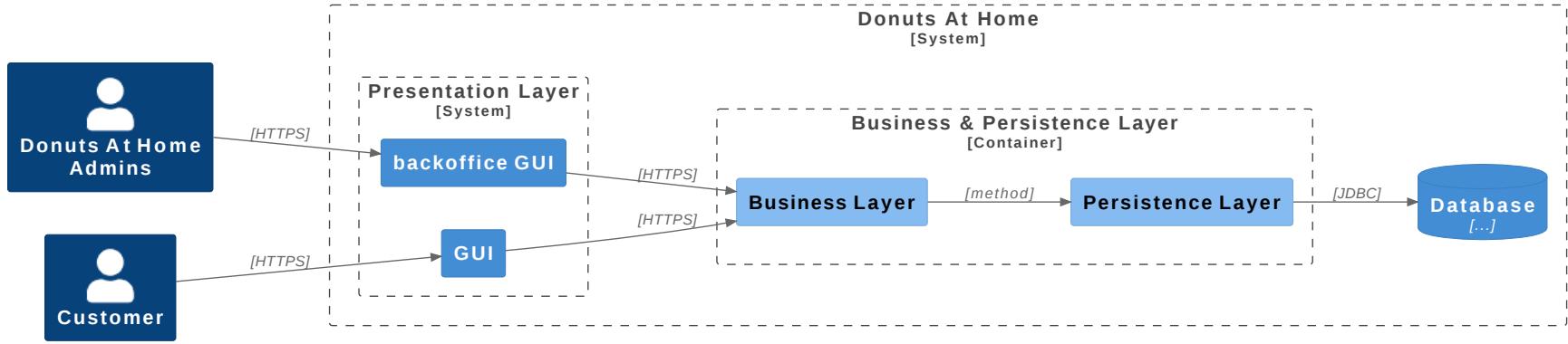
## Définition d' Amazon

*Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. These services are owned by small, self-contained teams.*

*Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.*

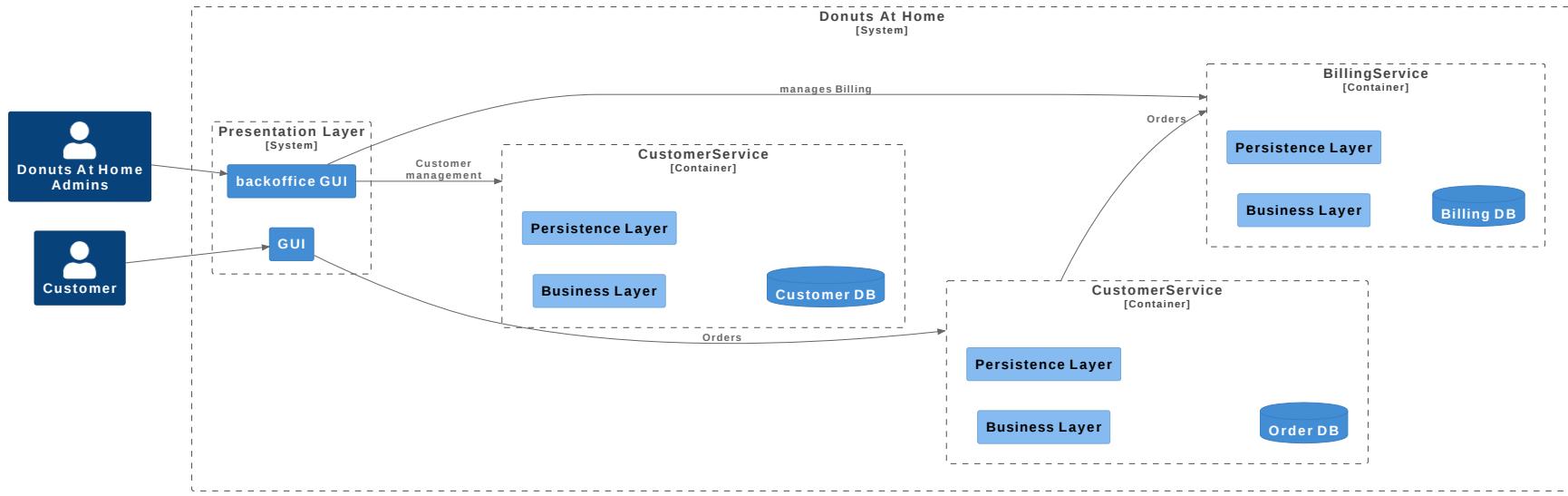
# Monolithe vs Microservices

## Un monolithe



Tous les composants sont déployés dans le même livrable et sur la même machine.

# Une plateforme microservices



Chaque domaine fonctionnel est déployé indépendamment.

# Pourquoi ?

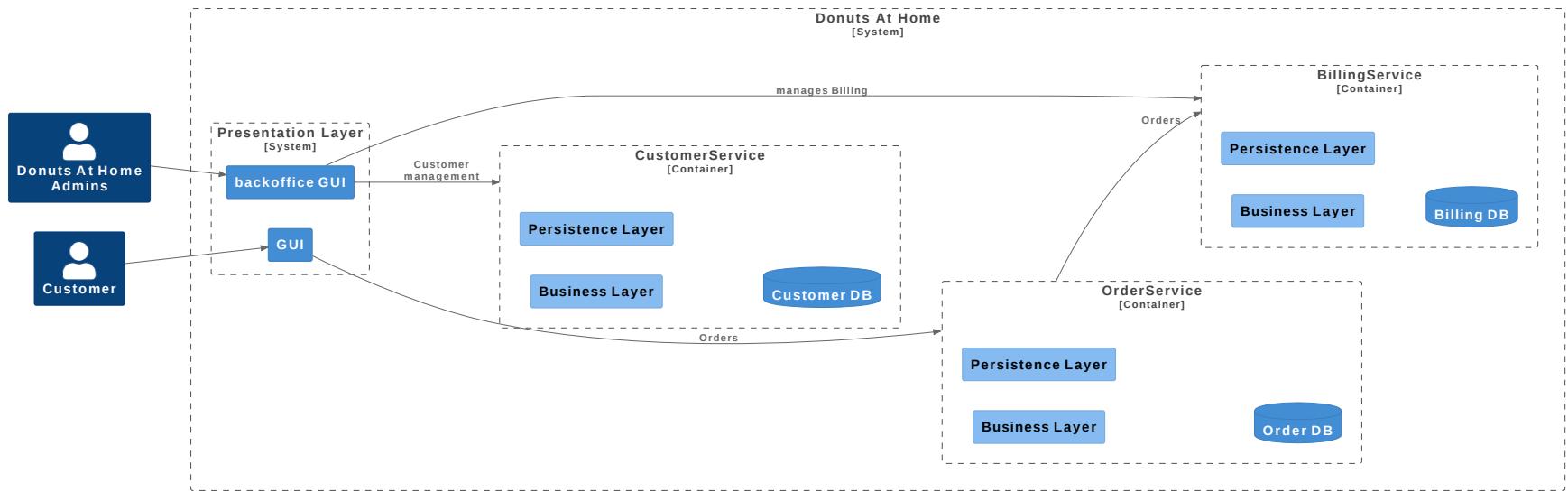
- Donner plus d'autonomie aux équipes en charge d'un microservice
- Faciliter la gestion du changement par des déploiements plus unitaires et modules (c.-à-d. plus petits)
- Favoriser l'innovation
- Utiliser la technologie la plus appropriée pour chaque service
- Améliorer et cibler la scalabilité

# Principes de base

Chaque microservice est composé d'une API et une base de données ou tout autre moyen de stockage (si nécessaire).

Il est développé et opéré par une "*Pizza Team*" qui fait tout du dev à la prod

Les fonctionnalités ne sont accessibles que par le biais des API --> On n'accède pas directement à la base de données.



# Protocoles



# Transactions synchrones

Deux grandes familles de protocole sont utilisées dans les transactions synchrones

Fonctionnalité	gRPC	API HTTP avec JSON
Contrat	Obligatoire (.proto)	Facultatif (OpenAPI)
Protocole	HTTP/2	HTTP
Payload	Protobuf (petit, binaire)	JSON (grand, lisible par l'homme)
Diffusion en continu	Client, serveur, bidirectionnel	Client, serveur
Prise en charge des navigateurs	Non (nécessite grpc-web)	Oui
Sécurité	Transport (TLS)	Transport (TLS)
Génération de code client	Oui	OpenAPI + outils tiers

# API HTTP

Les API basées sur HTTP utilisent soit GraphQL ou REST (majoritaire).

## Les API REST

Les API REST (Representational State Transfer) sont un style architectural pour la conception de services web. Elles utilisent les protocoles et les standards du web, principalement HTTP, pour permettre la communication entre un client et un serveur.

Voici les notions fondamentales concernant les API REST :

1. Ressources (ex. `/users/{userId}` )

2. Verbes HTTP:

- `GET` : Récupérer une ressource ou une collection de ressources. (ex. `/users/1` )
- `POST` : Créer une nouvelle ressource. (ex. Envoi d'un formulaire à la ressource `/users` )
- `PUT` : Mettre à jour une ressource existante.
- `DELETE` : Supprimer une ressource.
- `PATCH` : Mettre à jour une partie d'une ressource existante

3. Sans état (stateless)

4. Possibilité de mettre en cache les requêtes

5. Représentation (ex. JSON)

6. Interface uniforme (**c.-à-d. le contrat de services**)

7. Scalabilité et performance

# HATEOAS

HATEOAS (Hypermedia As The Engine Of Application State) est une contrainte de l'architecture REST selon laquelle un client interagit avec une application entièrement par le biais de ressources dynamiques fournies par le serveur. Ces ressources contiennent des liens hypermedia qui indiquent les actions possibles et les transitions d'état.

## **Caractéristiques de HATEOAS**

1. Auto-découverte
2. Navigation par liens
3. Dynamisme

## Exemple

Voici une API pour une boutique en ligne. Une réponse pour obtenir les détails d'un produit pourrait ressembler à ceci en utilisant JSON :

```
{
  "id": 123,
  "name": "Chaussures de course",
  "price": 50,
  "links": [
    {
      "rel": "self",
      "href": "http://api.boutique.com/products/123"
    },
    {
      "rel": "add-to-cart",
      "href": "http://api.boutique.com/cart/add/123"
    },
    {
      "rel": "reviews",
      "href": "http://api.boutique.com/products/123/reviews"
    }
  ]
}
```

# Modèle de maturité de Richardson

Ce modèle évalue la maturité d'une API REST en quatre niveaux (0 à 3), chacun ajoutant des caractéristiques supplémentaires qui rapprochent l'API des principes RESTful.

- **Niveau 0 :** Appels à distance uniques Une API avec une seule URL qui accepte différentes actions via des paramètres.
- **Niveau 1 :** Utilisation des ressources
  - `GET /users/getUser?id=123`
  - `POST /users/createUser`
- **Niveau 2 :** Verbes HTTP
  - `GET /users/123` (récupérer les détails de l'utilisateur avec l'ID 123)
  - `POST /users` (créer un nouvel utilisateur)
  - `PUT /users/123` (mettre à jour l'utilisateur avec l'ID 123)
  - `DELETE /users/123` (supprimer l'utilisateur avec l'ID 123)
- **Niveau 3 :** Hypermedia Controls

À ce niveau, l'API intègre des hyperliens dans les réponses pour guider les clients sur les actions possibles.

## Exemple de réponse HATEOAS

```
{  
  "id": 123,  
  "name": "Jane Doe",  
  "links": [  
    {  
      "rel": "self",  
      "href": "/users/123"  
    },  
    {  
      "rel": "friends",  
      "href": "/users/123/friends"  
    },  
    {  
      "rel": "update",  
      "href": "/users/123"  
    }  
  ]  
}
```

# GraphQL

GraphQL est un langage de requête pour les API. Il a été développé par Facebook en 2012 et open-source en 2015.

GraphQL permet aux clients de demander précisément les données dont ils ont besoin, et rien de plus

## Principes

### 1. Langage de requête

- GraphQL permet aux clients de définir la structure des réponses qu'ils souhaitent recevoir.
- Les requêtes GraphQL sont envoyées au serveur et le serveur retourne uniquement les données demandées.

### 2. Schéma fortement typé

### 3. Récupération de l'équivalent de plusieurs ressources en une seule requête

## Exemple de requête

```
{  
  user(id: "1") {  
    id  
    name  
    friends {  
      id  
      name  
    }  
  }  
}
```

## Exemple de réponse

```
{  
  "data": {  
    "user": {  
      "id": "1",  
      "name": "Alice",  
      "friends": [  
        {  
          "id": "2",  
          "name": "Bob"  
        },  
        {  
          "id": "3",  
          "name": "Charlie"  
        }  
      ]  
    }  
  }  
}
```

## En résumé

<b>Technologie</b>	<b>Avantages</b>	<b>Inconvénients</b>
REST	Simplicité de mise en oeuvre , Stateless, interopérabilité, utilisation des standards du Web	Manque de flexibilité dans les requêtes, gestion des versions, évolutivité, surcharge réseau
GrahQL	Flexibilité, efficacité, évolutivité, documentation automatique	Pas de cache possible, tout passe par une requête POST, complexité de mise en oeuvre

# Couplage

# Définition

Le couplage est une métrique indiquant le niveau d'interaction entre deux ou plusieurs composants logiciels (fonctions, modules, objets ou applications).

Deux composants sont dits couplés s'ils échangent de l'information. On parle de couplage fort ou couplage serré(fort) si les composants échangent beaucoup d'information.

On parle de couplage faible, couplage léger ou couplage lâche si les composants échangent peu d'information et/ou de manière désynchronisée.

Source: Wikipedia[0]

 **Une bonne architecture logicielle nécessite le couplage le plus faible possible!**

# Qu'est-ce qui crée un couplage dans les architectures microservices ?

## Couplage comportemental

- Description : Se produit lorsque des services partagent des responsabilités dans les processus métier.
- Conséquence : Si un service nécessite l'aide directe d'un autre pour accomplir ses tâches, cela indique que les périmètres des services n'ont pas été correctement définis.

## Couplage des connaissances

- Description : Se produit lorsque les services connaissent trop bien les implémentations internes des autres.

## Couplage des schémas

- Description : Se produit lorsque les services sont liés à un ensemble commun d'interfaces ou de schémas.

## Couplage temporel

- Description : Se produit lorsqu'un service attend une réponse immédiate d'un autre avant de pouvoir continuer.

## Couplage des processus

- Description : Se produit lorsque les services commencent à assumer trop de responsabilités distinctes.
- Conséquence : Donne lieu à des implémentations de services difficiles à mettre à l'échelle ou à modifier.

## Couplage d'implémentation

- Description : Se produit lorsque les services partagent des détails d'implémentation plutôt que des contrats ou des schémas.
- Conséquence : Les API qui laissent échapper des détails d'implémentation.

## Couplage basé sur la localisation :

- Description : Se produit lorsqu'un service s'attend à ce qu'une ressource existe à un emplacement spécifique.

# L'anti pattern: le plat de spaghetti

En informatique, le syndrome du plat de spaghetti est une dégradation qui touche les systèmes informatiques trop fortement couplés. Le système devient coûteux à maintenir et sujet aux pannes.

## Principales causes

1. Couplage fort entre les services
2. Flux de communication désorganisés
3. Absence de séparation claire des responsabilités

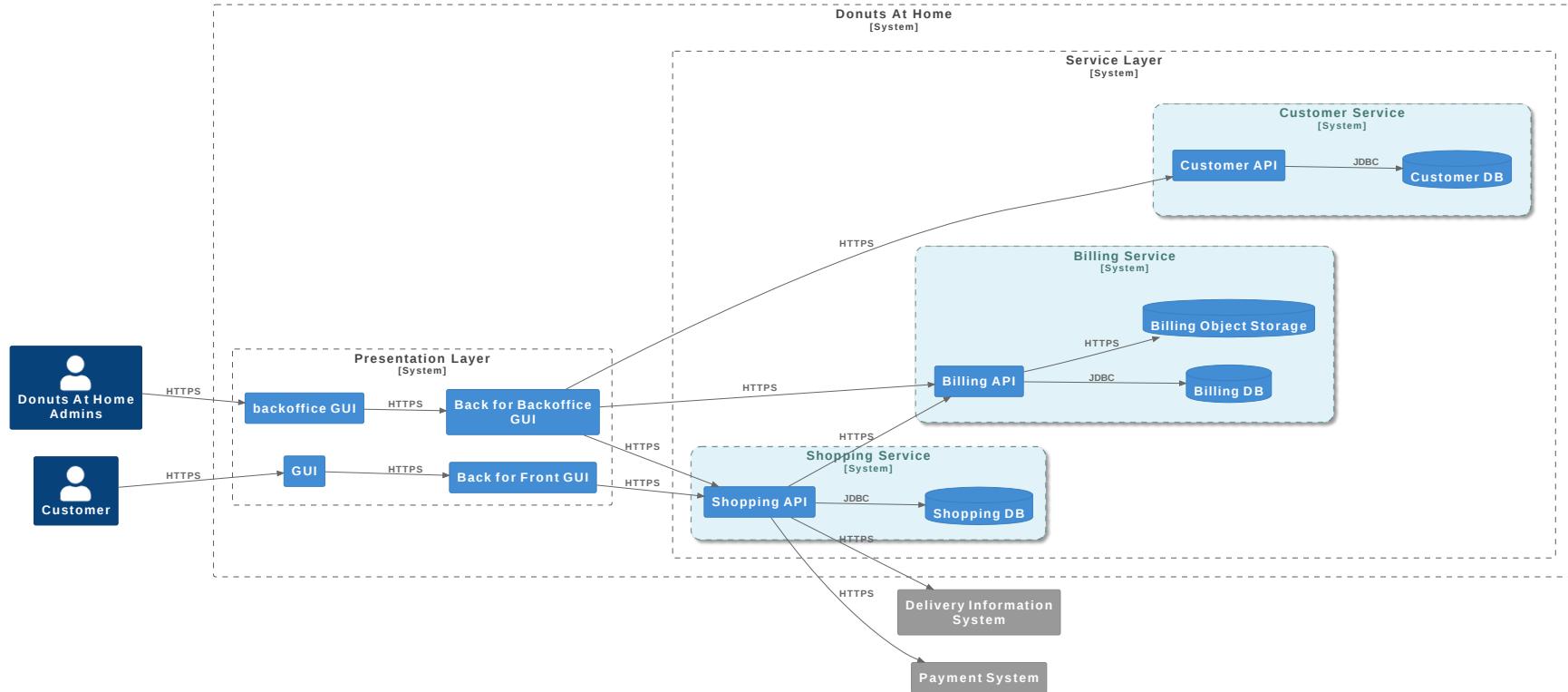
## Conséquences du *Spaghetti Code*

1. Maintenance complexe
2. Difficulté à tester
3. Difficulté à analyser les erreurs en production
4. Impossibilité de *scalier*

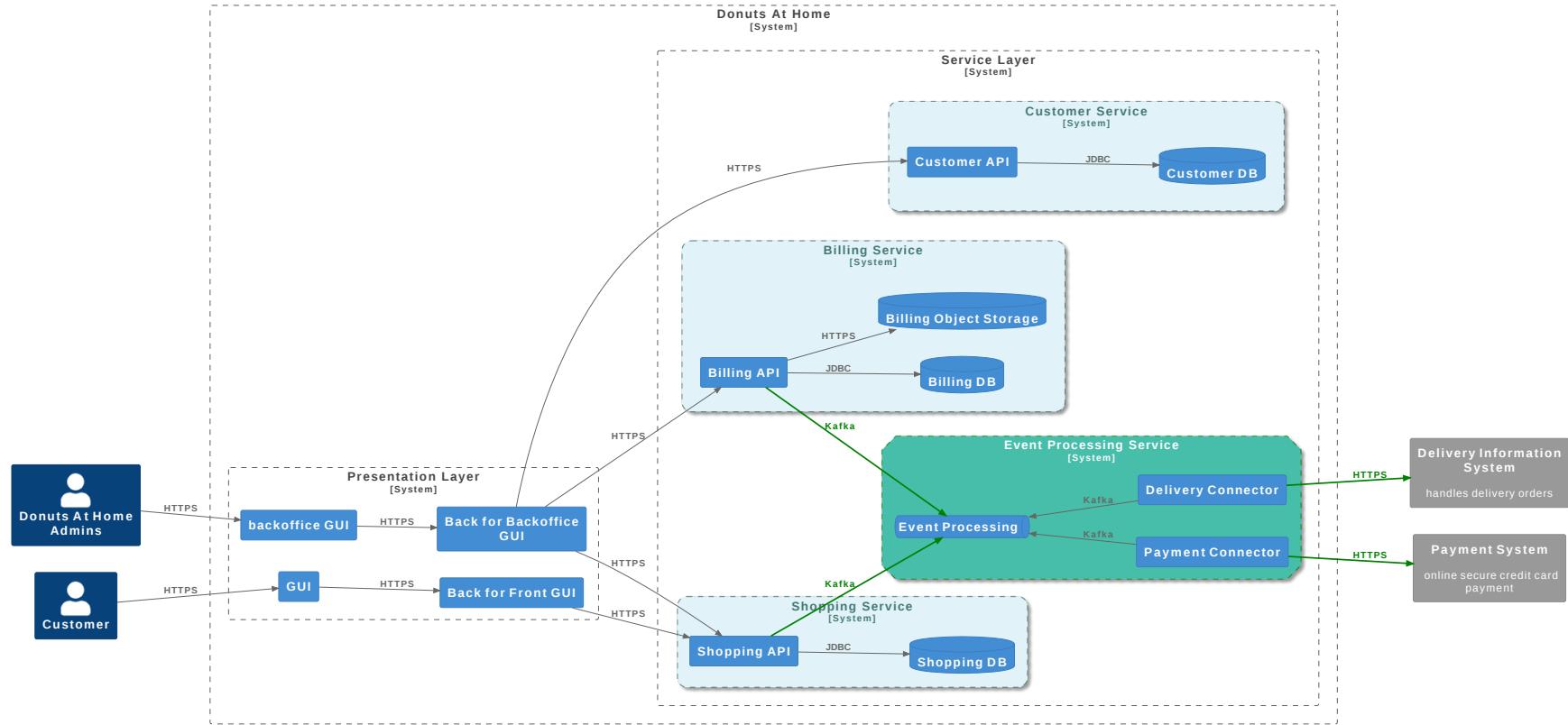
## Comment l'éviter ?

1. Définir les limites claire (*Bounded Context*[1])
2. Utilisation des messages asynchrones
3. Prise en compte de la résilience dès la conception (ex. *Circuit Breaker*[2])
4. Isolation des données (c.-à-d. 1 base de données par microservice)
5. Automatiser les tests et déploiements pour déployer et valider chaque service indépendamment.

# Exemple de couplage fort



# Comment découpler les appels externes ?



# Messageries asynchrones

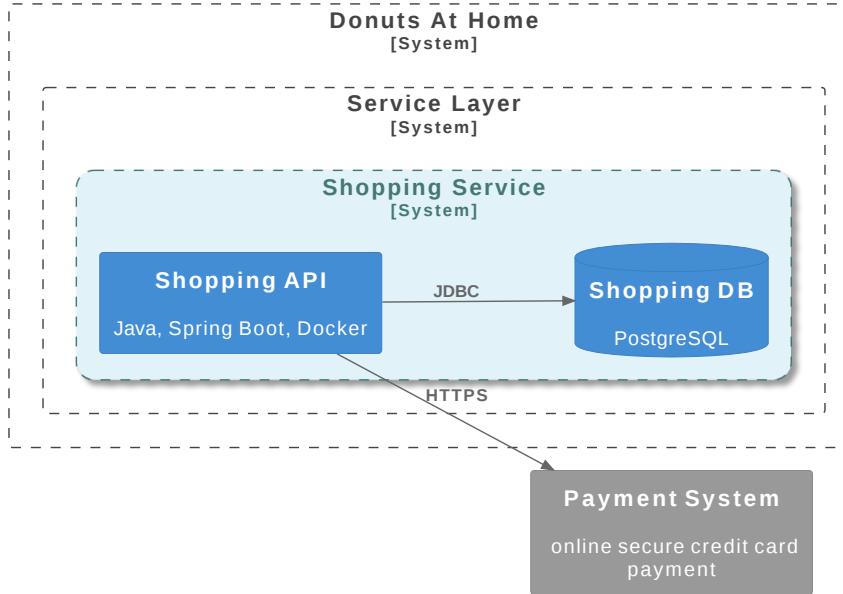
Les architectures microservices offrent leur capacité à décomposer les applications en services indépendants et faiblement couplés. La communication entre les services est cruciale pour le fonctionnement harmonieux de l'ensemble du système.

Deux approches principales existent pour orchestrer cette communication : les messageries synchrones (API) et asynchrones.

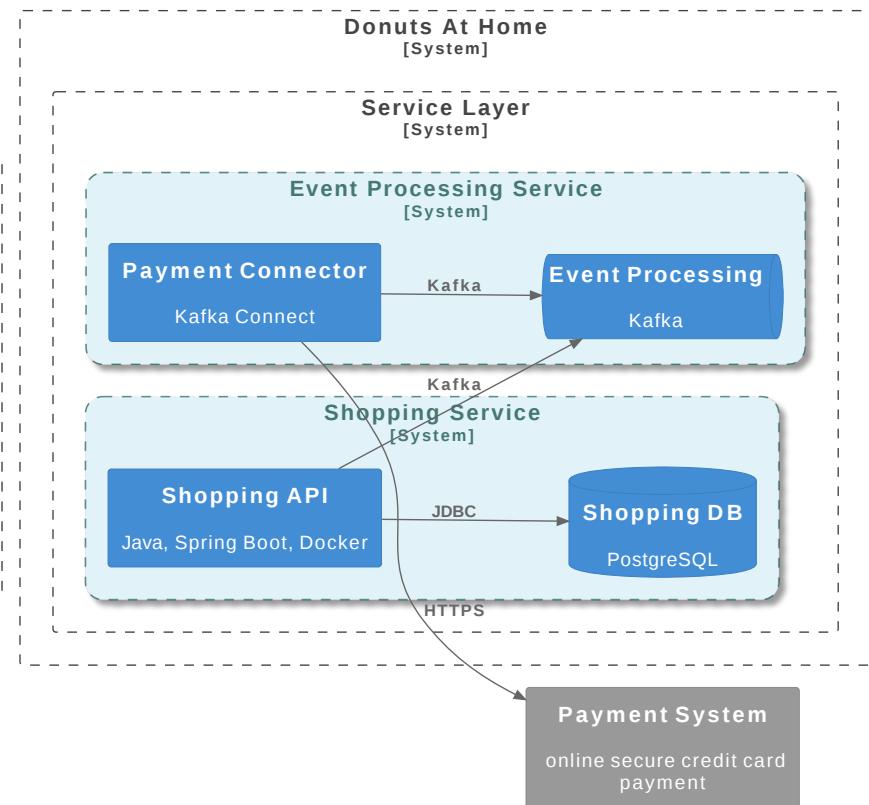
Ces dernières améliorent la résilience, la scalabilité et la flexibilité des architectures microservices.

# Qu'est-ce qu'une messagerie asynchrone ?

## Transaction synchrone



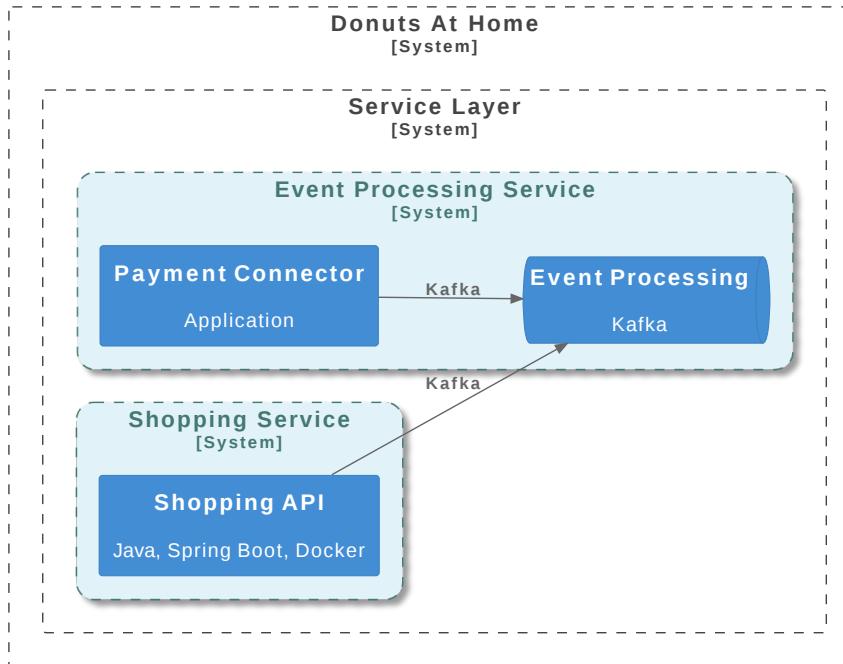
## Transaction asynchrone



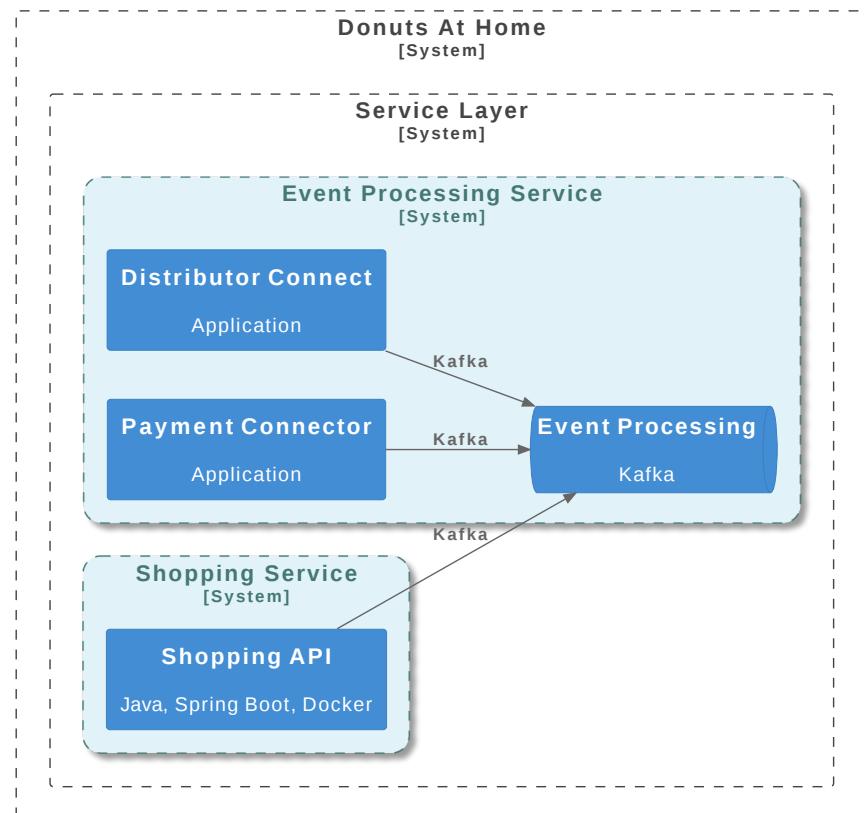


# Modèles de communication

## Point-à-Point



# Publish/Subscribe (Pub/Sub)



## Avantages

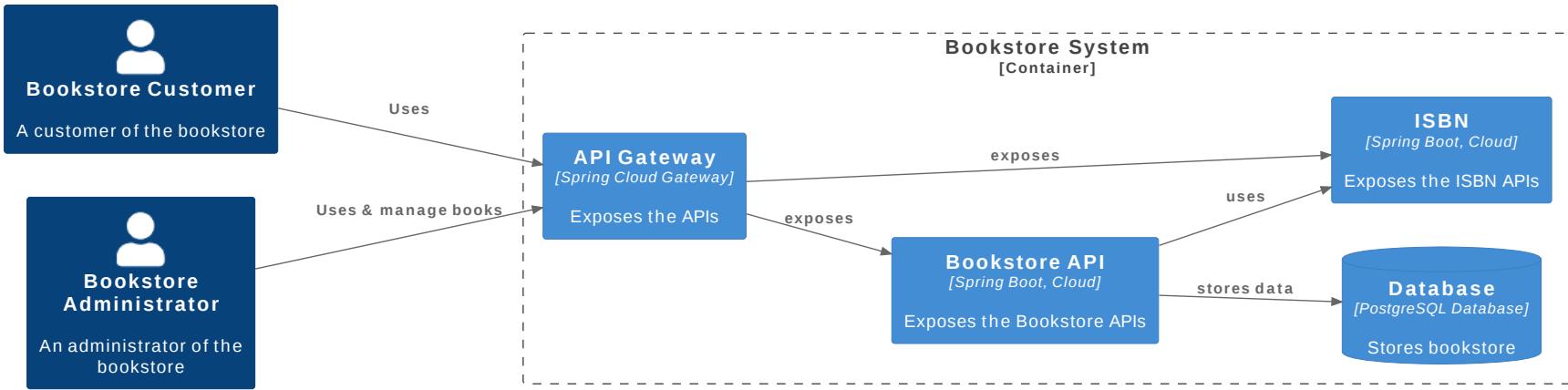
- Scalabilité améliorée
- Augmente la résilience des plateformes microservices
- Traitement au fil de l'eau des événements

## Inconvénients

- La consistence des données est compliquée à mettre en oeuvre
- La gestion des erreurs est plus compliquée
- Il faut superviser le système de messagerie et gérer les cas où la file d'attente est saturée



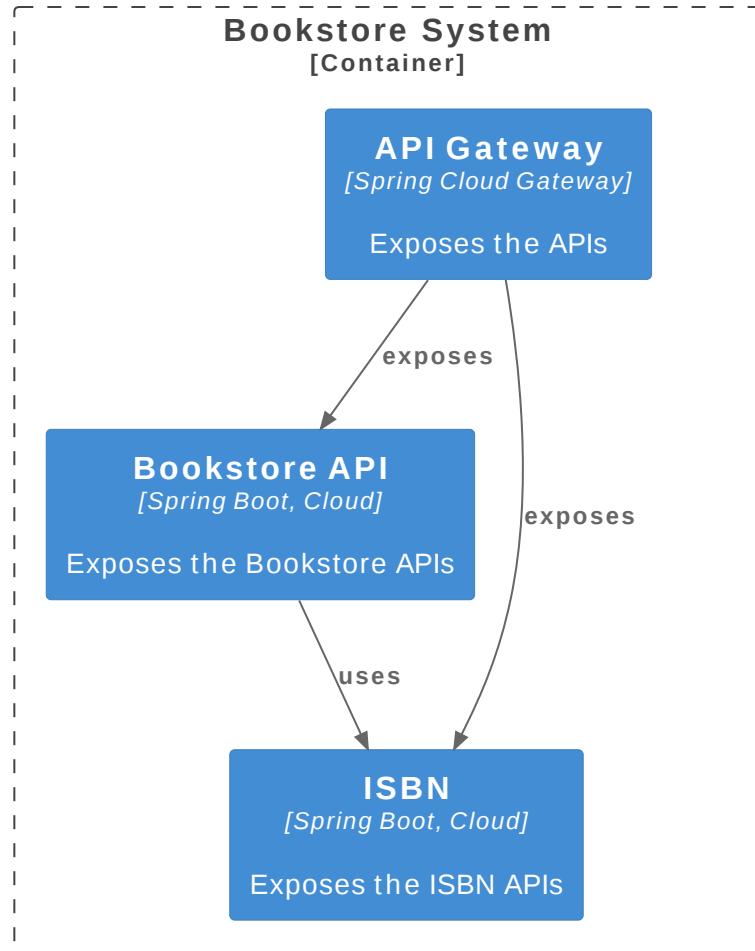
# L' API Gateway



Exemple chez NetFlix[3]

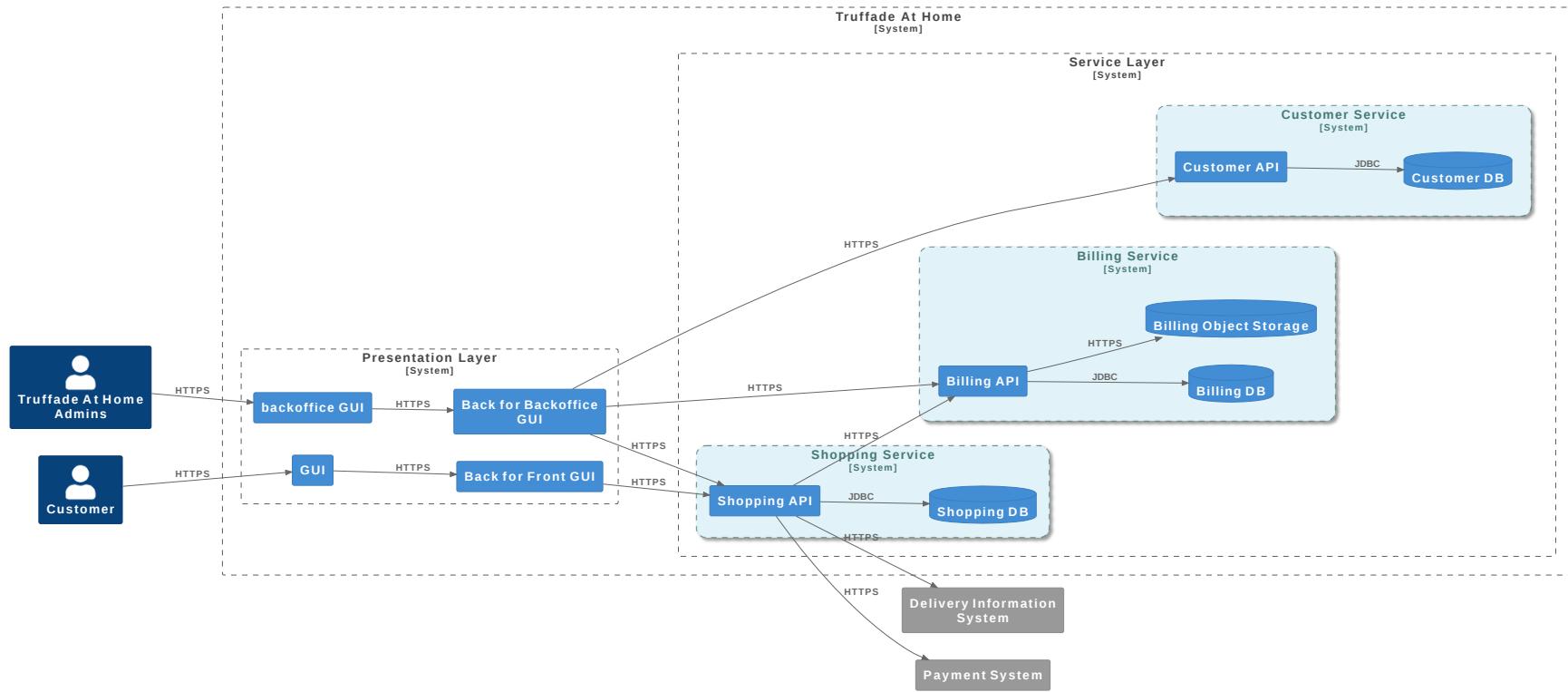
# Principes clés d'une API Gateway

1. Gestion Centralisée des API
2. Routage
3. Sécurité
4. Transformation des Requêtes et des Réponses
5. Rate Limiting et Throttling
6. Caching
7. Surveillance et Journalisation
8. Haute Disponibilité et Scalabilité
9. Gestion des Versions d'API
10. Gestion des Erreurs
11. Compatibilité Multi-protocoles



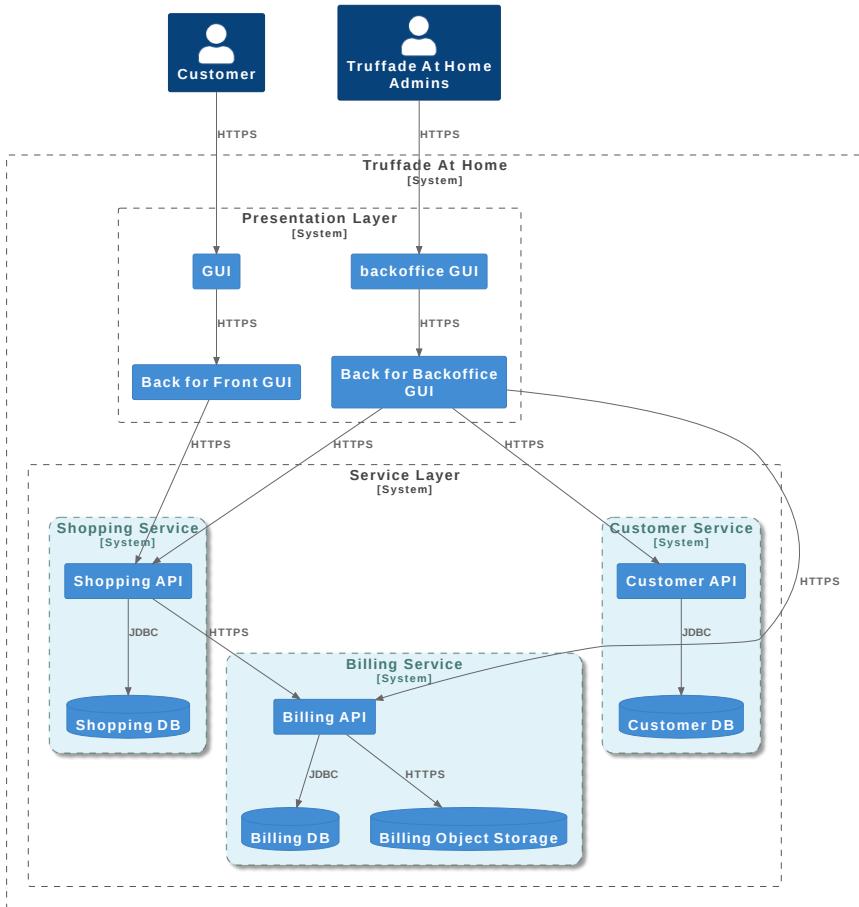
# Back For Front (BFF)

Un exemple:



# Principes Clés d'un BFF

1. Personnalisation des API
2. Séparation des Préoccupations (*Separation of concerns*)
3. Réduction de la Complexité du Frontend
4. Optimisation des Performances
5. Transformation des Données
6. Gestion des Échecs et des Erreurs
7. Agilité et Flexibilité
8. Sécurité
9. Gestion des Dépendances
10. Réutilisation

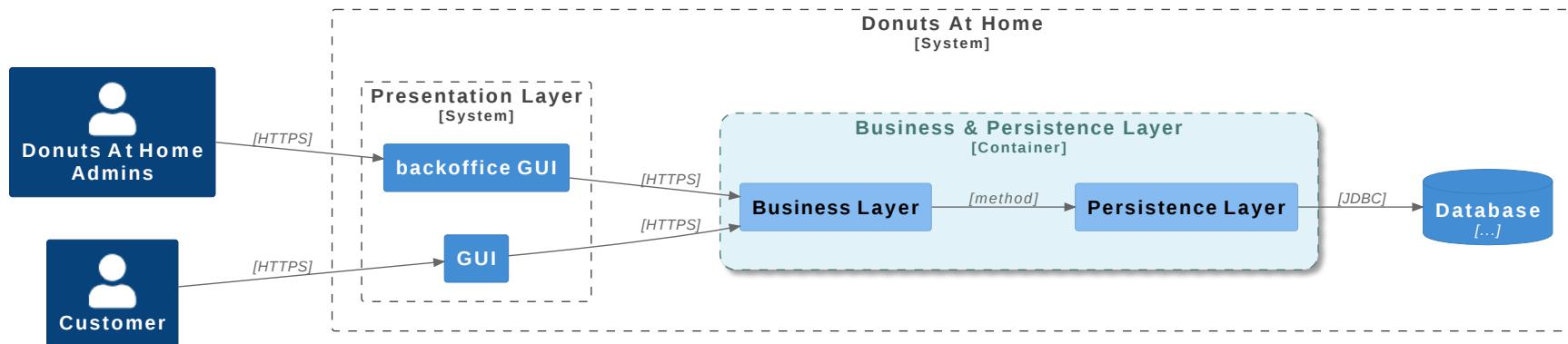


# Transactions distribuées

# Rappels : ACID

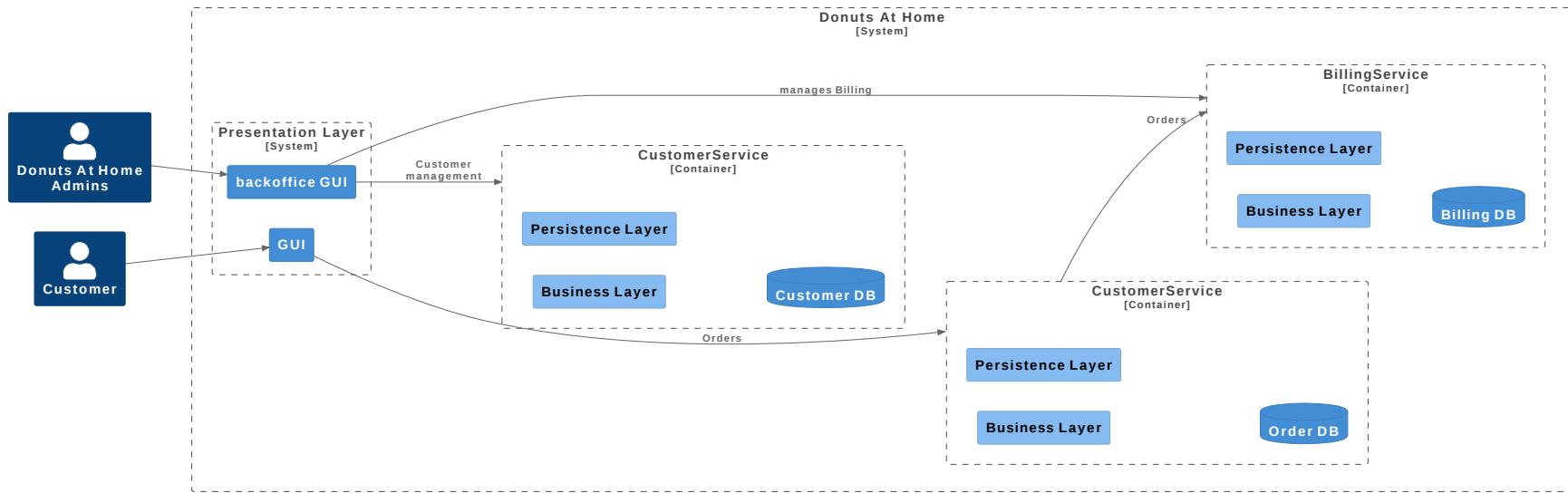
- Atomicité
- Cohérence
- Isolation
- Durabilité

Source: Wikipedia [4]



# Comment faire dans une architecture distribuée ?

## Exemple



Si dans l'appel à `BillingService` échoue:

- Comment traiter les erreurs et garantir la cohérence des données dans `CustomerService` ?
- Comment piloter une transaction de bout en bout ?

# Les Long Running Actions

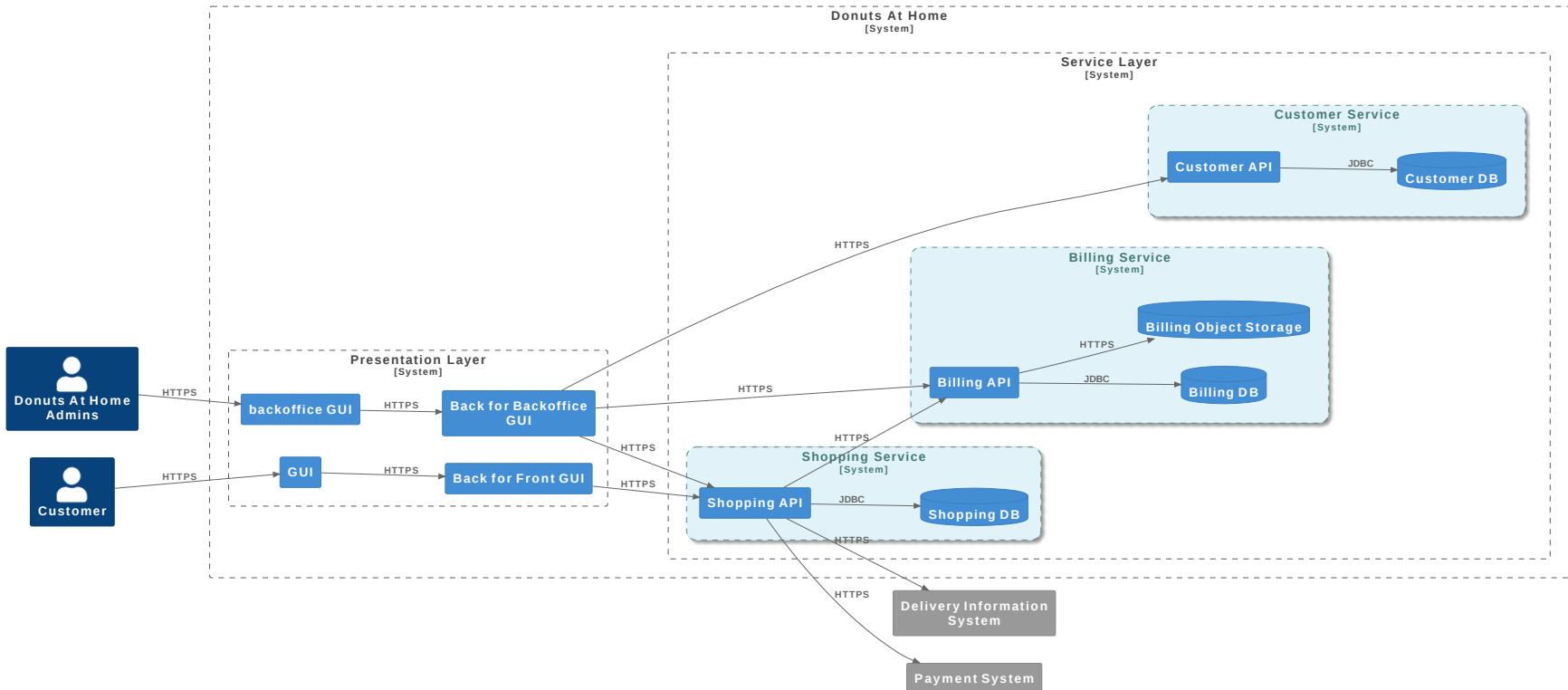
## Définition

Les "long running actions" dans le contexte des microservices sont des opérations qui durent longtemps, souvent beaucoup plus que la durée typique d'une requête HTTP.

Ces opérations prolongées peuvent inclure des traitements complexes, des calculs intensifs, la génération de rapports, l'intégration avec des systèmes externes, la migration de données, etc.

Gérer ces actions efficacement est crucial pour maintenir la performance, la scalabilité et la résilience de l'ensemble du système de microservices.

# Exemple



## Caractéristiques

Durée prolongée

Complexité

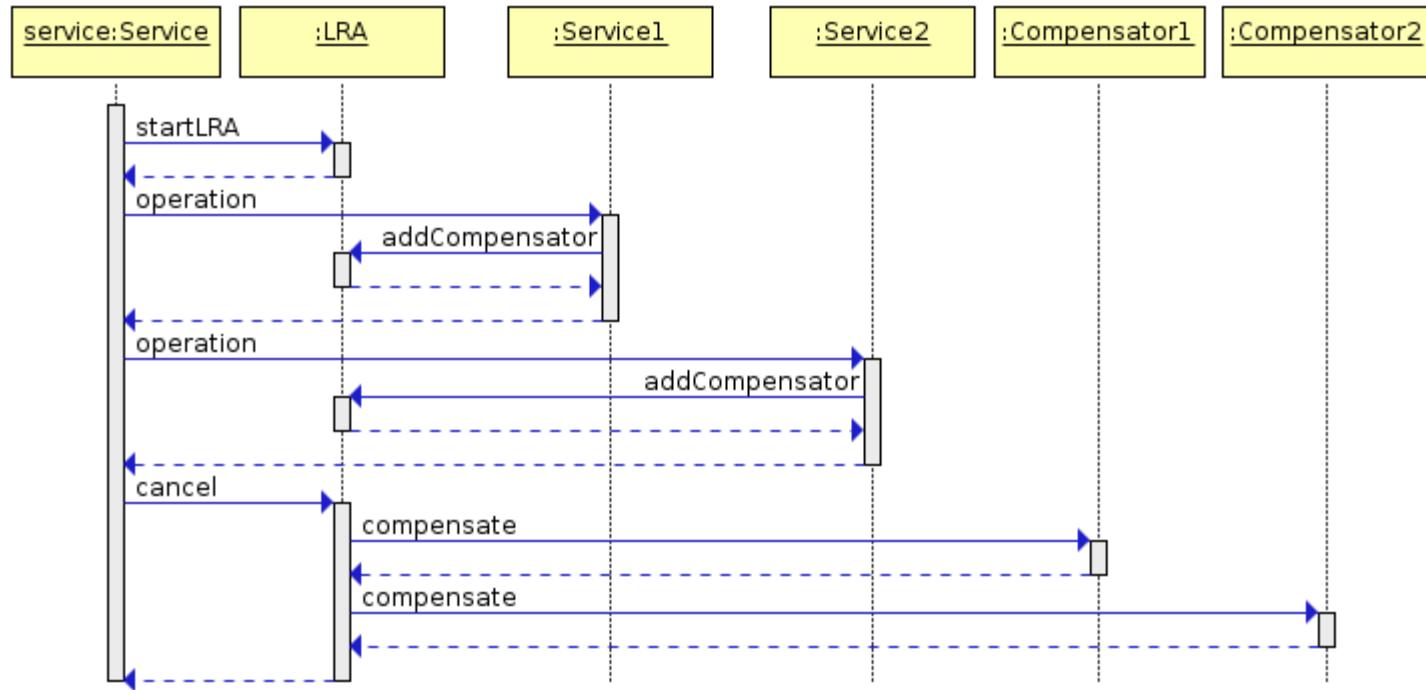
État Intermédiaire

Scalabilité

Tolérance aux Erreurs

# Les Long Running Actions en Java

## Modèle



- Article présentant les LRA

# Le pattern SAGA

# L'observabilité

L'observabilité est la capacité à comprendre l'état interne d'un système complexe. Lorsqu'un système est observable, un utilisateur peut identifier la cause première d'un problème de performance en examinant les données qu'il produit, sans tests ou codage supplémentaires. C'est l'un des moyens par lesquels les problèmes de qualité de service peuvent être abordés.

# Logs, Traces et Monitoring

# Les logs

Les logs (ou traces applicatives) sont des enregistrements chronologiques d'événements survenus dans une application ou un système.

Ils servent à fournir des informations détaillées sur les actions exécutées, les erreurs rencontrées et l'état du système à différents moments.

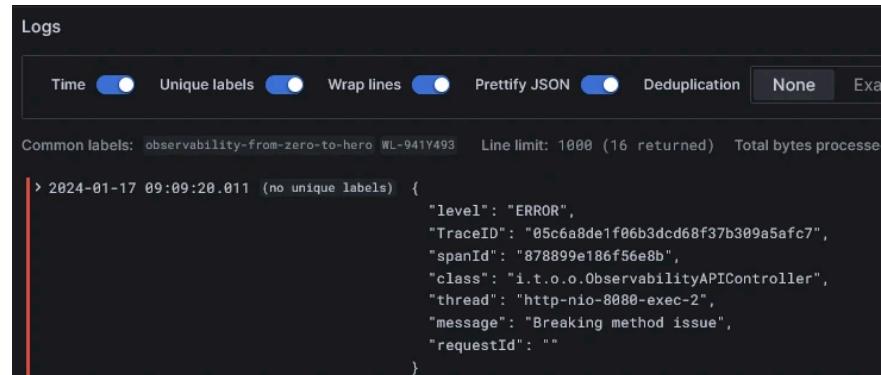
Quel log est utile ?

```
System.out.println("Je suis ici...");
```

ou

```
LOG.warn("Check card expiry date does not pass: bad format: {
```

Un exemple de log dans un format structuré:



The screenshot shows a log viewer with the following interface elements:

- Header: "Logs"
- Control buttons: Time (on), Unique labels (on), Wrap lines (on), Prettify JSON (on), Deduplication (on), None (selected), Examples (disabled).
- Statistics: Common labels: observability-from-zero-to-hero WL-941Y493, Line limit: 1000 (16 returned), Total bytes processed: 1.00 MB.
- Log entry:

```
> 2024-01-17 09:09:20.011 (no unique labels) {
    "level": "ERROR",
    "TraceID": "05c6a8de1f06b3dc68f37b309a5afc7",
    "spanId": "878899e186f56e8b",
    "class": "i.t.o.o.ObservabilityAPIController",
    "thread": "http-nio-8080-exec-2",
    "message": "Breaking method issue",
    "requestId": ""
}
```

A red vertical bar highlights the timestamp "2024-01-17 09:09:20.011".

## Principes clés

- Distinguer les logs de développement (utilisés pour le débogage) des logs de production.
- Utiliser les niveaux de log appropriés ( DEBUG , INFO , WARN , ERROR , etc.).
- Fournir des messages clairs et utiles pour les OPS.
- Inclure le contexte métier pertinent (par exemple, l'échec de la création du contrat 123456).
- Assurer que les logs puissent être lus par un outil externe (comme un agrégateur de logs).
- Ne pas exposer de données sensibles dans les logs en respectant les normes de confidentialité telles que GDPR et PCI DSS.

# Les agrégateurs de logs

Google Cloud fra-wlcp-product-dev Search (/) for resources, docs, products, and more Search

Logs Explorer Refine scope Project Share link Learn

Query Recent (17) Saved (0) Suggested (17) Library Clear query Save Stream logs Run query

Last 1 hour Search all fields All resources All log names All severities Show query

Log fields Timeline Create metric Create alert Jump to now More actions

Search fields and values

SERVICE

- wl-system 54,082
- wlcp-product-demo 16,201
- kube-system 10,784
- wlcp-product-edge 9,417
- wlcp-product-dev 4,437
- kubecost 2,830
- goldilocks 2,489
- wlcp-product-iacc 1,799
- forecasting 1,436
- cost-analyzer 1,380

435,444 results SEVERITY TIME CEST ↑ SUMMARY Edit Summary fields Wrap lines

2024-09-23 16:09:28.824 "HTTP" verb="GET" URI="/readyz?exclude=kms-provider-&exclude=kms-provider-1&timeout=2s" latency="576.907us" userAgent="gke-master-healthcheck" audit-ID="f9dcf61d-4a66-43b2-8d0c-047574607cbc" srcIP="10.128.0.1" dstIP="10.128.0.1" method="GET" io.k8s.coordination.v1.leases.update /e-system/leases/cloud-controller-manager system:cloud-controller-m... audit\_log, method: "io.k8s.coordination.v1.leases.update" 2024-09-23 16:09:28.827 "HTTP" verb="PUT" URI="/apis/coordination.k8s.io/v1/namespaces/kube-system/leases/cloud-controller-manager?timeout=5s" latency="6.974853ms" userAgent="cloud-controller-manager/v1.28.2 (linux/amd64)" io.k8s.coordination.v1.leases.get /espaces/kube-system/leases/addon-manager system:addon-manager audit\_log, method: "io.k8s.coordination.v1.leases.get", principal\_email="system:anonymous" 2024-09-23 16:09:28.894 "HTTP" verb="PUT" URI="/apis/coordination.k8s.io/v1/namespaces/kube-system/leases/addon-manager" latency="1.939268ms" userAgent="addon-manager/1.29-v0.19 (673f330)" audit-ID="0b88c356-550d-4f66-84f... 2024-09-23 16:09:28.899 "HTTP" verb="PUT" URI="/apis/coordination.k8s.io/v1/namespaces/kube-system/leases/addon-manager" latency="4.061621ms" userAgent="addon-manager/1.29-v0.19 (673f330)" audit-ID="7287d853-0296-4cd6-88d... 2024-09-23 16:09:28.943 "HTTP" verb="WATCH" URI="/apis/networking.gke.io/v1alpha1/redirectservices?allowWatchBookmarks=true&resourceVersion=2059899152&timeout=7m28s&timeoutSeconds=448&watch=true" latency="7m28.001737537s" ... 2024-09-23 16:09:28.945 "Starting watch" path="/apis/networking.gke.io/v1alpha1/redirectservices" resourceVersion="205985364" labels="" fields="" timeout="8m8s" 2024-09-23 16:09:28.967 "gke-cluster-dev" k8s.io io.k8s.get readyz system:anonymous audit\_log, method: "io.k8s.get", principal\_email: "system:anonymous" 2024-09-23 16:09:28.967 "HTTP" verb="GET" URI="/readyz" latency="1.94835ms" userAgent="kube-probe/1.29" audit-ID="9cf73c5a-060a-4d8e-be0e-1338b25d0ffe" srcIP="127.0.0.1:34300" apf\_pl="exempt" apf\_fs="probes" apf\_lsseats=1 ... 2024-09-23 16:09:29.000 "HTTP" verb="WATCH" URI="/apis/batch/v1/cronjobs?allowWatchBookmarks=true&resourceVersion=205899466&timeout=7m7s&timeoutSeconds=427&watch=true" latency="7m7.001748503s" userAgent="vpa-recommender/v... 2024-09-23 16:09:29.002 "Starting watch" path="/apis/batch/v1/cronjobs" resourceVersion="205980359" labels="" fields="" timeout="7m27s" 2024-09-23 16:09:29.020 "gke-cluster-dev" k8s.io io.k8s.coordination.v1.leases.get /cloud-provider-extraction-migration-pt2 system:cloud-controller-m... audit\_log, method: "io.k8s.coordination.v1.leases.get", principal\_email="system:anonymous" 2024-09-23 16:09:29.020 "HTTP" verb="GET" URI="/apis/coordination.k8s.io/v1/namespaces/kube-system/leases/cloud-provider-extraction-migration-pt2?timeout=5s" latency="2.353239ms" userAgent="cloud-controller-manager/v1.28.2 (linux/amd64)" io.k8s.coordination.v1.leases.update /Cloud-provider-extraction-migration-pt2 system:cloud-controller-m... audit\_log, method: "io.k8s.coordination.v1.leases.update" 2024-09-23 16:09:29.021 "HTTP" verb="WATCH" URI="/apis/networking.gke.io/v1alpha1/remotenodes?allowWatchBookmarks=true&resourceVersion=2059899095&timeout=5m9s&timeoutSeconds=319&watch=true" latency="5m19.001303312s" userAgent="vpa-recommender/v... 2024-09-23 16:09:29.023 "Starting watch" path="/apis/networking.gke.io/v1alpha1/remotenodes" resourceVersion="205985366" labels="" fields="" timeout="6m27s" 2024-09-23 16:09:29.026 "gke-cluster-dev" k8s.io io.k8s.coordination.v1.leases.update /Cloud-provider-extraction-migration-pt2 system:cloud-controller-m... audit\_log, method: "io.k8s.coordination.v1.leases.update"

RESOURCE TYPE

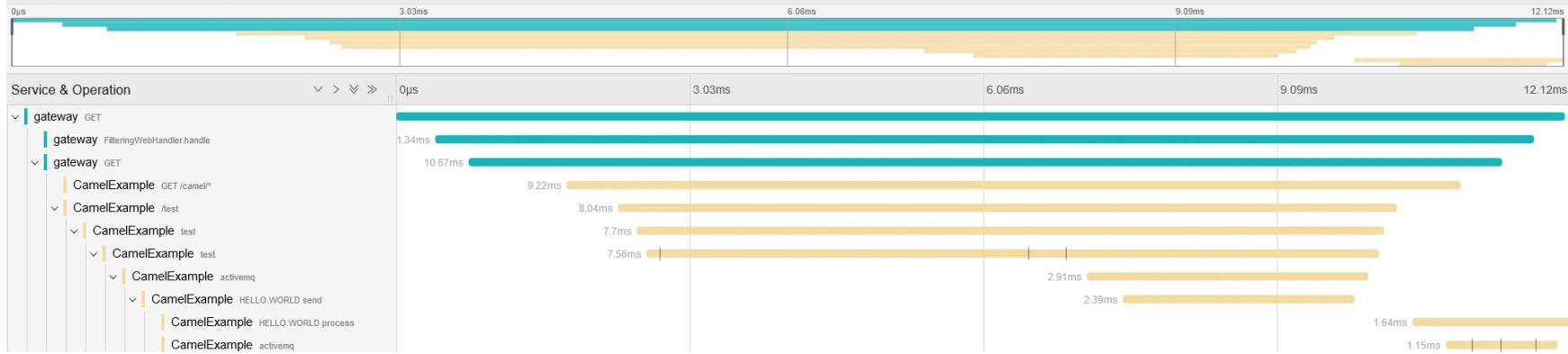
- Kubernetes Control Plane Com... 153,691
- Kubernetes Cluster 149,770
- Kubernetes Container 102,088
- GKE Cluster Operations 15,150

## Les traces distribuées

1. Suivi des transactions de bout en bout
2. Corrélation des logs
3. Détection et isolation des problèmes
4. Contexte global
5. Optimisation de la Performance

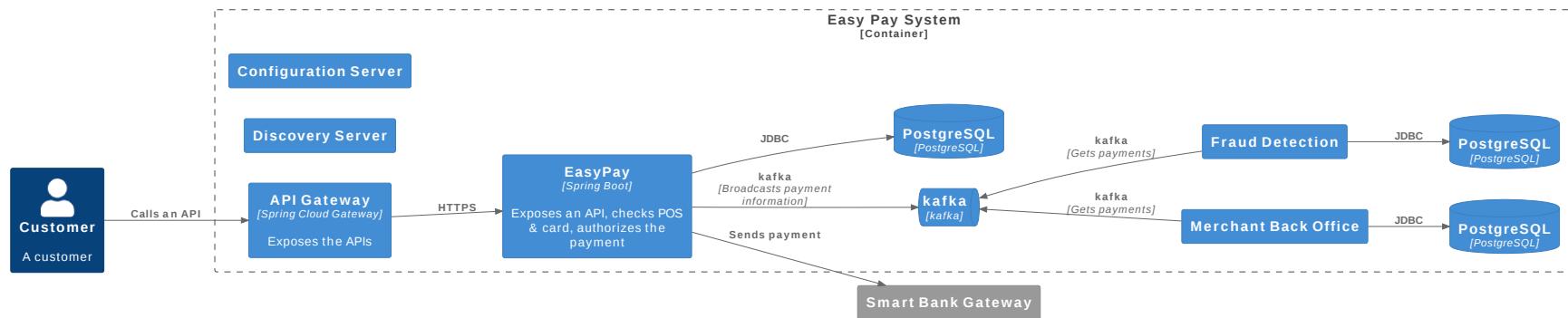
Attention aux performances!

Trace Start August 31 2023, 17:44:30.155 Duration 12.12ms Services 2 Depth 8 Total Spans 11



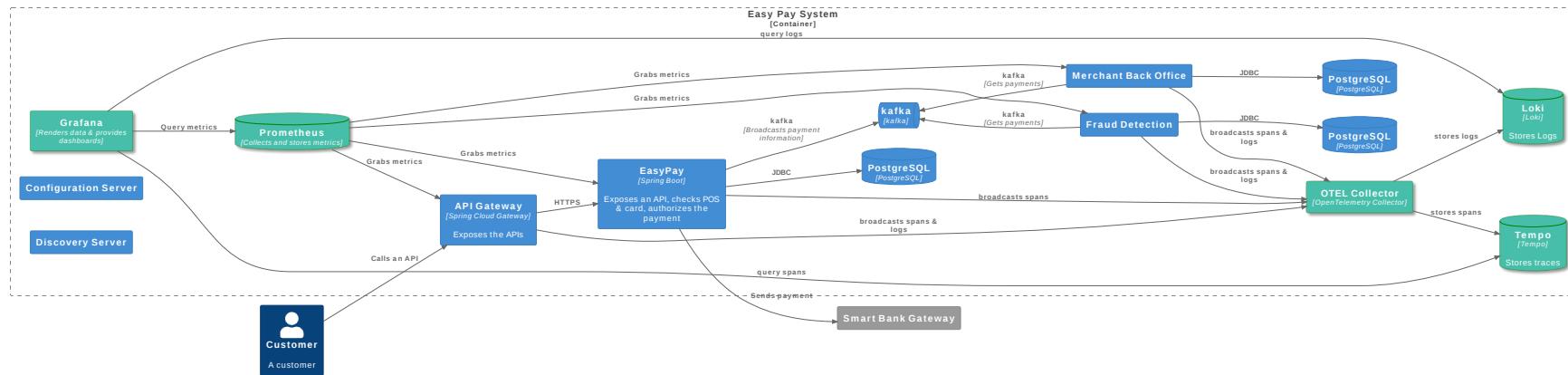
# Un exemple d'implémentation

Une application de paiement basée sur des microservices

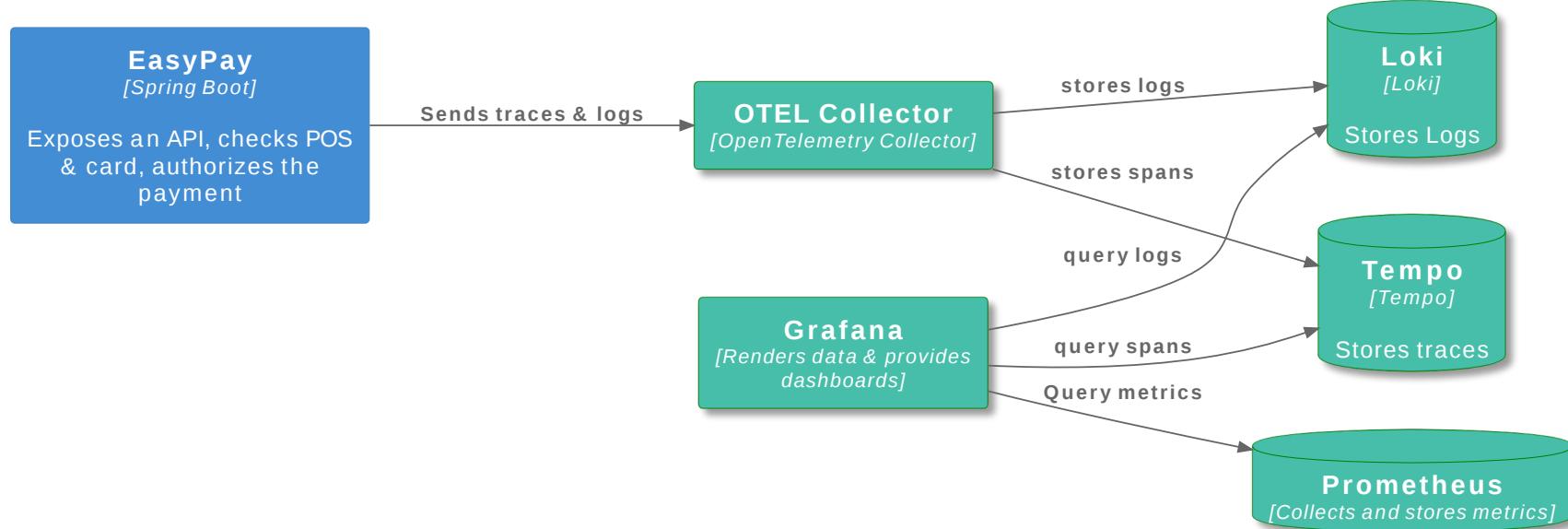


## Utilisation de la plateforme Grafana

- Loki[6] va stocker les logs
- Tempo[7] va stocker les traces
- OTEL Collector[8] va collecter les logs et traces
- Grafana[9] va permettre l'affichage et la recherche



## Zoom sur un service



# Sécurité des architectures microservices

# Principaux défis

- Décentralisation
- Multiplicité des points d'accès et d'exposition
- Complexité des communications inter-services sécurisées
- Gestion des secrets et des configurations sécurisées
- Sécurisation des données en transit et au repos

# Authentification

L'authentification est le processus de vérification de l'identité d'un utilisateur ou d'un service pour s'assurer qu'il est bien celui qu'il prétend être.

Dans une architecture microservices, chaque requête doit être authentifiée avant d'accéder aux ressources pour éviter l'accès non autorisé.

Protocoles: OpenID Connect (#RefErr)

# Autorisation

L'autorisation est le processus de contrôle d'accès aux ressources ou actions en fonction des droits d'un utilisateur ou d'un service.

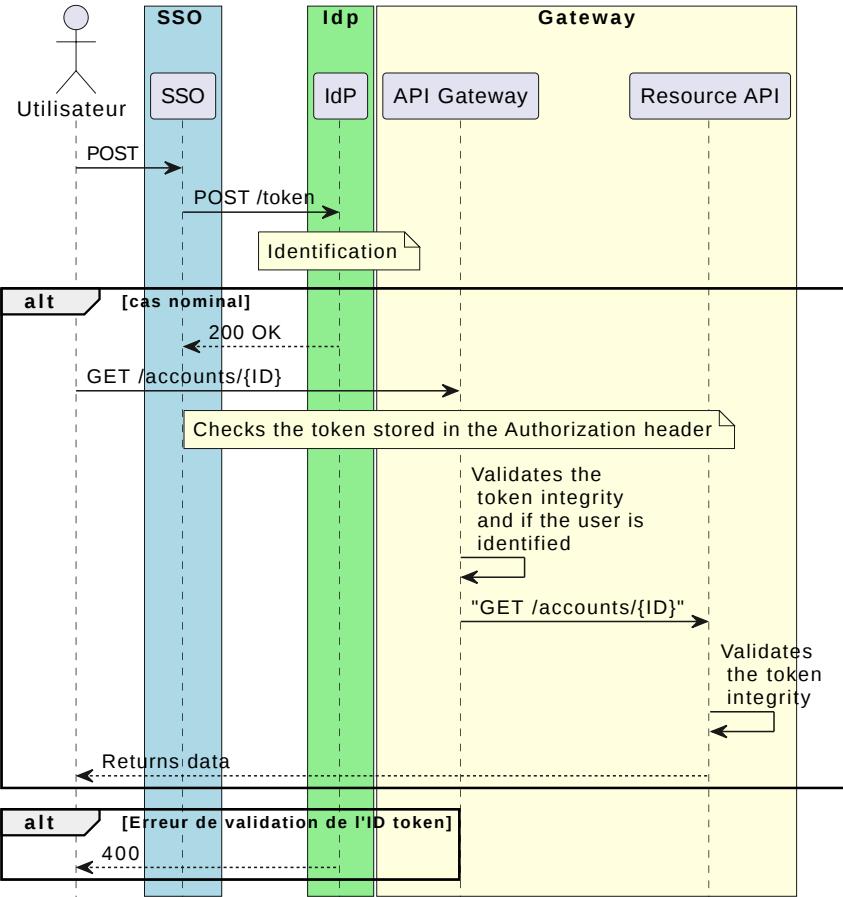
Après l'authentification, l'autorisation permet de déterminer les actions qu'un utilisateur est habilité à effectuer.

Dans une architecture microservices, chaque service doit vérifier l'autorisation de l'utilisateur pour garantir un accès sécurisé et approprié.

Protocoles: OAuthv2

# Authentification avec JSON Web Token (JWT)

1. Connexion de l'utilisateur
2. Vérification des identifiants
3. Création d'un jeton
4. Envoi du fichier à l'utilisateur
5. Vérification et utilisation du jeton dans toutes les requêtes avec l'en-tête `Authorization: Bearer <token>`



# Anatomie d'un jeton JWT

Les JWT sont définis par la RFC 7519[10] Ils sont composés de trois parties :

- Header : Contient des informations sur le type de token (JWT) et l'algorithme de signature utilisé.
- Payload : Contient les informations ou claims de l'utilisateur (ex., ID, rôle, permissions).
- Signature : Une signature générée avec la clé secrète, qui garantit l'intégrité du token.

La version encodée :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkw  
bmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF  
4fwpMeJf36P0k6yJV_adQssw5c
```

Pour valider des JWT : <https://jwt.io/>

## L'en-tête

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

## Le Payload

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxN
```

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

## La signature

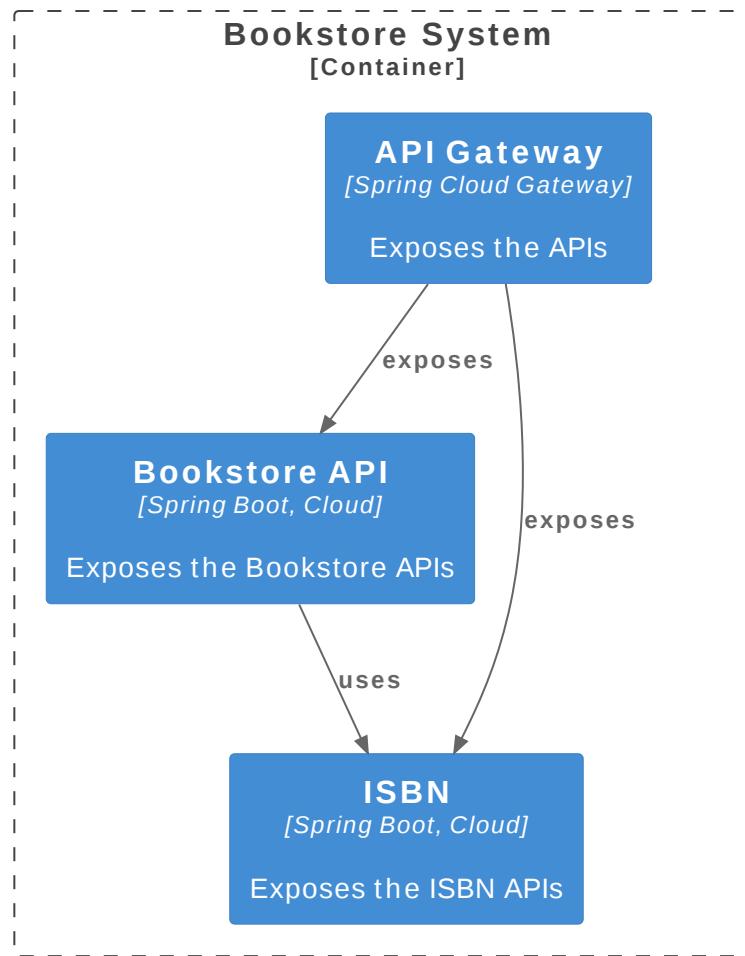
```
SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

## Pour aller plus loin

- [OAUTH 2.1 expliqué simplement \(même si tu n'es pas dev\) ! \(Julien Topçu\)](#)
- [Spécification OAuthv2](#)
- [Site officiel OpenID Connect](#)

# API Gateway et sécurité des communications

- **API Gateway** : Elle sert de point d'entrée unique, permettant de gérer l'authentification, l'autorisation, la limitation de débit (rate limiting), et d'appliquer des règles de sécurité.
  - **Chiffrement des communications** : Assurer que toutes les communications entre les microservices et avec les clients soient chiffrées (TLS/SSL).
  - **Zero Trust Architecture [11]** : Bien que la gateway aie validé le jeton, les services sous-jacents doivent valider les droits pour chaque API.
- 💡 **Bonnes pratiques** : Utiliser HTTPS pour toutes les communications et configurer des certificats SSL pour chaque microservice.



# Sécurité des données en transit et au repos

- **Données en transit** : Chiffrer les données en transit entre microservices et avec les clients pour protéger les informations sensibles. Le chiffrement des canaux de communication (HTTPS, TLS) empêche les attaques d'interception.
  - **Données au repos** : Chiffrer les bases de données et les systèmes de stockage pour éviter le vol de données en cas de compromission. Utiliser des clés de chiffrement robustes et les protéger avec des solutions de gestion de clés.
- 💡 **Bonnes pratiques** : Mettre en œuvre une gestion centralisée des clés et chiffrer systématiquement toutes les données sensibles au repos.

# Contrôle des accès réseau et pare-feu

- **Contrôle des accès réseau** : Restreindre les accès réseau entre les microservices pour limiter les chemins d'accès potentiels en cas de compromission. Configurer des réseaux privés virtuels (VPN) pour les communications internes.
  - **Pare-feu et segmentations** : Utiliser des pare-feux et des règles de segmentation pour limiter la visibilité des microservices entre eux et depuis l'extérieur. Une segmentation réseau stricte empêche un attaquant de se déplacer latéralement dans l'architecture.
- ⌚ **Bonnes pratiques** : Mettre en œuvre des contrôles réseau de type *zero trust* pour vérifier chaque communication entre microservices.

En conclusion

Si vous voulez aller plus loin



# Ressources sur le web

- <https://microservices.io/>
- <https://aws.amazon.com/fr/microservices/>
- <https://www.f5.com/company/blog/nginx/microservices-at-netflix-architectural-best-practices>
- <https://www.youtube.com/watch?v=7HC6ZfSy8M4>

# Quelques conseils

- Les architectures micro services peuvent répondre à différents problèmes techniques et organisationnels
- Ce n'est pas une *Silver Bullet*. Les monolithes peuvent très bien convenir pour un grand nombre de besoins
- Attention à la complexité !

# Bibliography

- 1. Wikipedia. Couplage. Available from: [`https://fr.wikipedia.org/wiki/Couplage\_\(informatique\)`](https://fr.wikipedia.org/wiki/Couplage_(informatique)) (23)
- 2. Fowler M. Bounded Context. Available from: [`https://martinfowler.com/bliki/BoundedContext.html`](https://martinfowler.com/bliki/BoundedContext.html) (27)
- 3. Wikipedia. Circuit Breaker. Available from: [`https://en.wikipedia.org/wiki/Circuit\_breaker\_design\_pattern`](https://en.wikipedia.org/wiki/Circuit_breaker_design_pattern) (27)
- 4. Netflix. Utilisation de l'API Gateway chez Netflix. Available from:  
[`https://netflixtechblog.com/optimizing-the-netflix-api-5c9ac715cf19`](https://netflixtechblog.com/optimizing-the-netflix-api-5c9ac715cf19) (37)
- 5. Wikipedia. Propriétés ACID. Available from:  
[`https://fr.wikipedia.org/wiki/Propri%C3%A9t%C3%A9s\_ACID`](https://fr.wikipedia.org/wiki/Propri%C3%A9t%C3%A9s_ACID) (42)
- 6. Richardson C. Saga Pattern. Available from: [`https://microservices.io/patterns/data/saga.html`](https://microservices.io/patterns/data/saga.html) (47)
- 7. Grafana. Grafana Loki. Available from: [`https://grafana.com/oss/loki/`](https://grafana.com/oss/loki/) (56)
- 8. Grafana. Grafana Tempo. Available from: [`https://grafana.com/oss/loki/`](https://grafana.com/oss/loki/) (56)
- 9. OpenTelemetry. OpenTelemetry Collector. Available from: [`https://opentelemetry.io/`](https://opentelemetry.io/) (56)
- 10. Grafana. Grafana Stack. Available from: [`https://grafana.com/about/grafana-stack/`](https://grafana.com/about/grafana-stack/) (56)

Fin

Introduction aux architectures micro services  
Alexandre Touret

 Creative Commons Attribution 4.0 International