

# Introduction aux architectures micro services

Cours Architectures distribuées  
Alexandre Touret

# 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
- Cohérence des données, transactions distribuées et utilisation de messageries asynchrones
- Les bases de données
- Gestion des pannes et erreurs
- La sécurité
- Comment casser un monolithe ?

# Fondamentaux

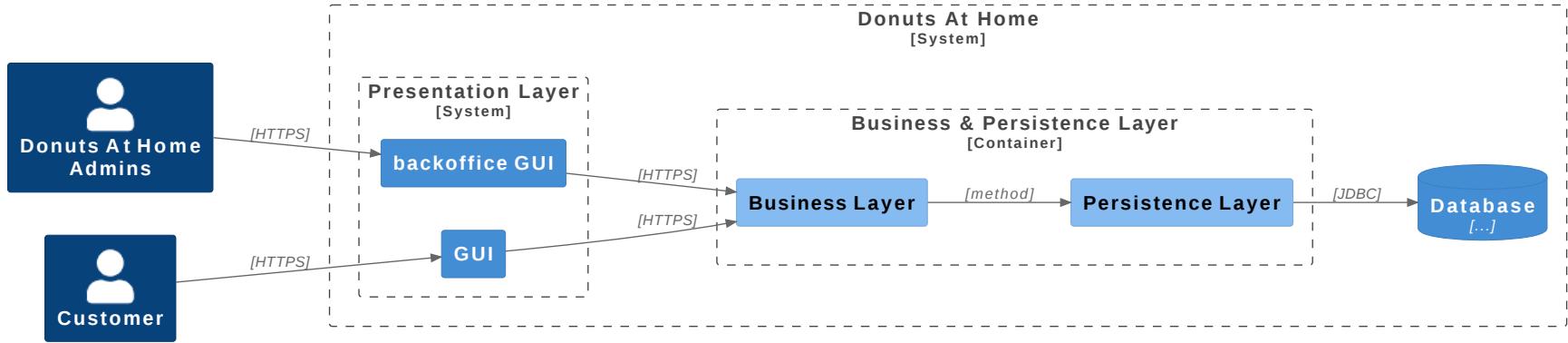
## Définition d' Amazon

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.

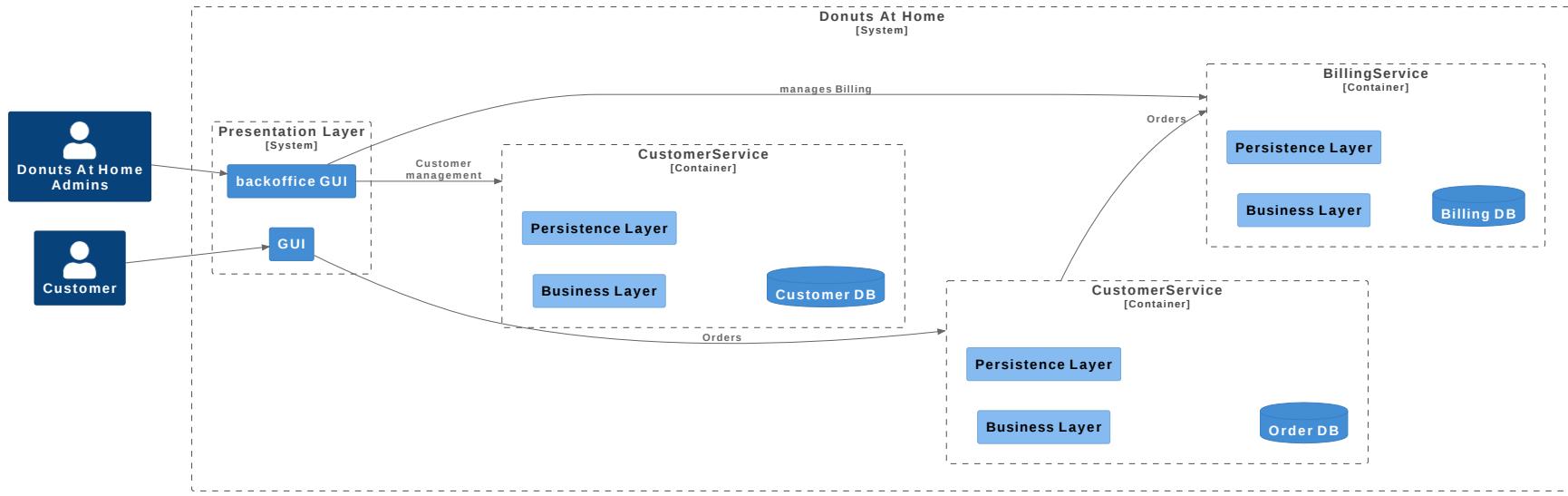
# Monolithe vs Microservices

## 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 ?

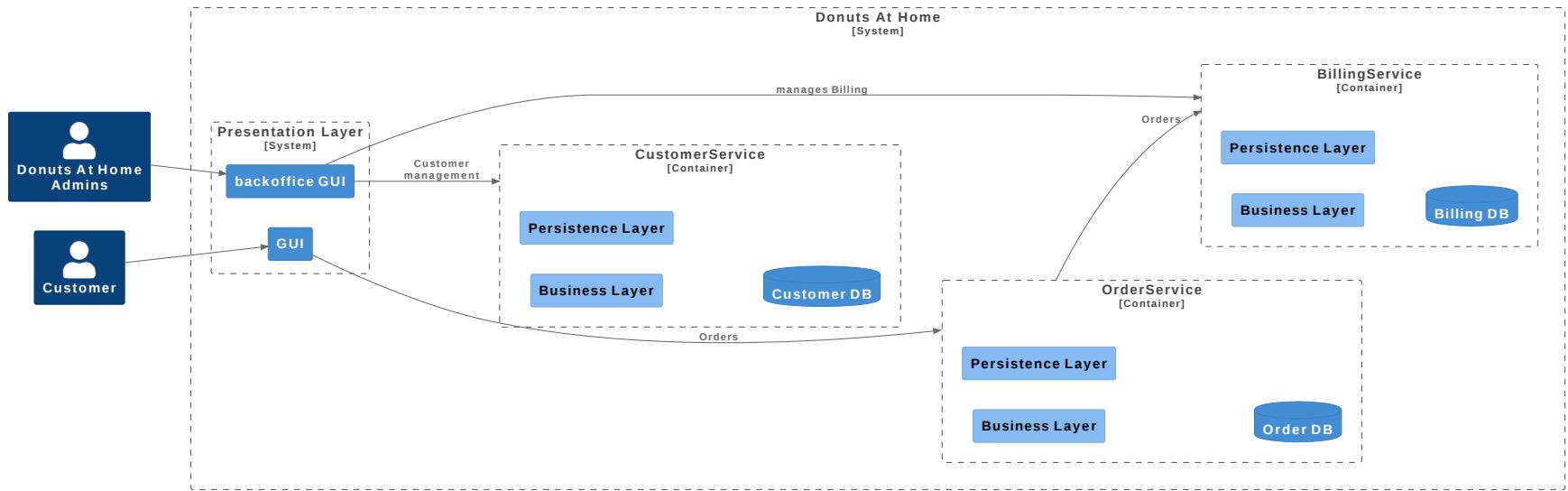
- Donner plus d'autonomie aux équipes en charge d'un microservice
- Faciliter la gestion du changement par des déploiements plus unitaires et modules (c.-à-d. plus petits)
- Favoriser l'innovation
- Utiliser la technologie la plus appropriée pour chaque service
- Améliorer et cibler la scalabilité

# 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

Il y a deux types de flux utilisés dans les connexions entre services :

- Synchrones: Utilisation de protocoles basés sur HTTP ou GRPC
- Asynchrones: Utilisation de messageries asynchrones ou streaming de données

# 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 quelques points clés 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

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.

- 1. Niveau 0 : Appels à distance uniques** Exemple :  
Une API avec une seule URL qui accepte différentes actions via des paramètres.

- 2. Niveau 1 : Utilisation des ressources** Exemple :

- GET /users/getUser?id=123
- POST /users/createUser

- 3. Niveau 2 : Verbes HTTP** Exemple :

- GET /users/123 (récupérer les détails de l'utilisateur avec l'ID 123)
- POST /users (créer un nouvel utilisateur)

## 4. 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

Technologie	Avantages	Inconvénients
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



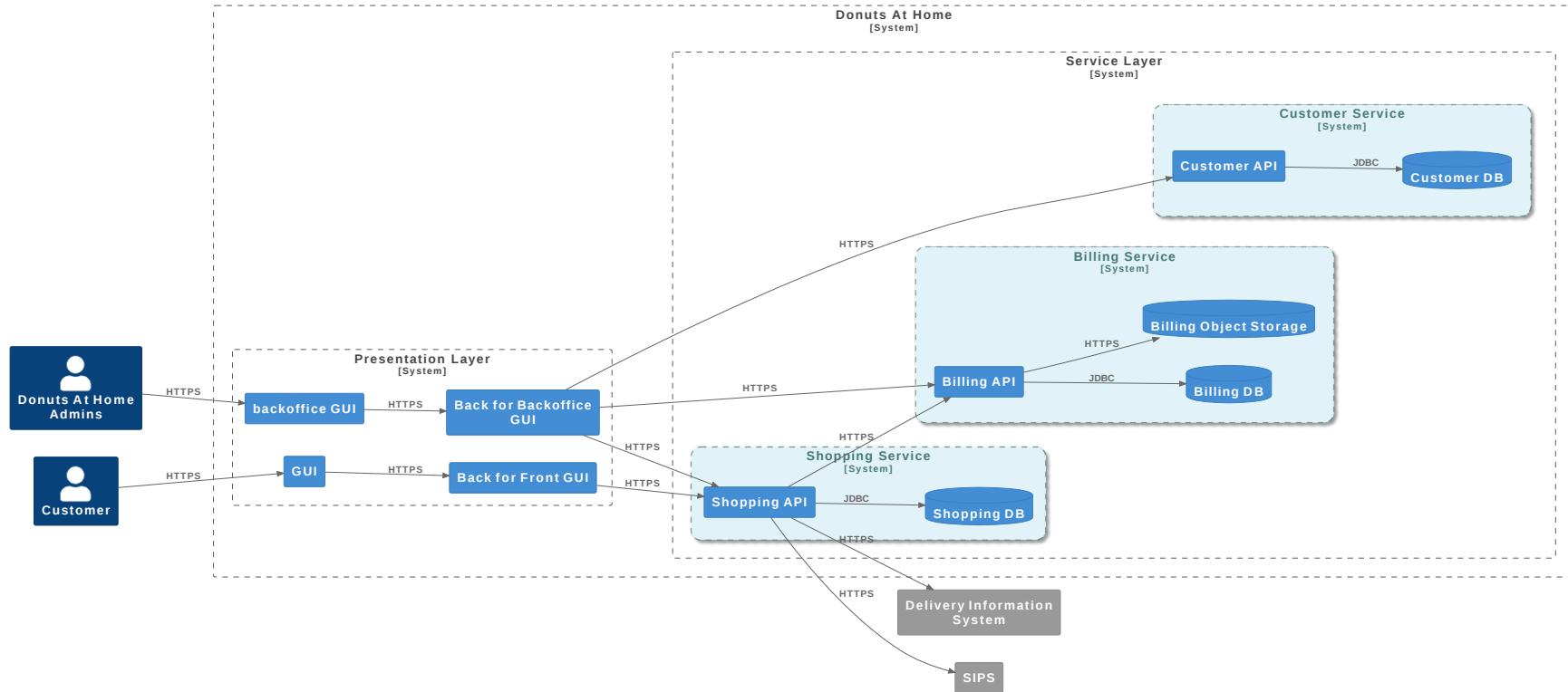
# Couplage lâche

Le couplage fait référence au degré d'interdépendance entre les différents modules, composants ou classes d'un système logiciel. Un couplage fort signifie que les modules sont fortement interconnectés et dépendent étroitement les uns des autres, tandis qu'un couplage faible indique une interdépendance minimale entre les modules. Le couplage est une mesure importante de la qualité du design logiciel, car un couplage fort peut rendre le système difficile à maintenir, à modifier et à tester. En revanche, un couplage faible favorise la modularité, la réutilisabilité et la maintenabilité du code.

# Qu'est-ce qui crée du couplage dans les architecturs microservices ?

1. Couplage comportemental
2. Couplage des connaissances
3. Couplage temporel
4. Couplage d'implémentation
5. Couplage basé sur la localisation

# Exemple de couplage fort



# Comment découpler les appels externes ?

