



# Les APIs Restful

---

David THIBAU - 2022

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

- **Introduction**

- Écosystème et enjeux
- Fondations des APIs RestFul
- SOAP vs REST
- Les alternatives à REST

- **Définition d'APIs**

- Recommandations pragmatiques
- Conventions
- Gestion du versionning
- Bonnes pratiques de conception
- Open API Specification
- Consumer Driven Contract

- **Design et outils**

- Les outils autour de OpenAPI 3.0
- Le projet Spring Cloud Contract
- Le rôle d'une Gateway
- API Management

- **Implémentation avec la POO**

- Architecture logicielle d'un service
- Les patterns pour implémenter la logique métier
- Couche contrôleur
- Les problématiques de sérialisation/désérialisation
- Impacts sur la couche DAO
- Exception, CORS

- **Sécurité**

- OWASP et vulnérabilités
- Authentification et autorisations
- OpenID/oAuth2 et JWT



# Introduction

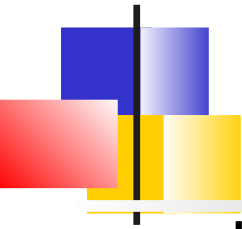
---

## **Eco-système et enjeux**

Fondations

SOAP vs REST

Alternatives à SOAP et REST



# L'écosystème moderne

---

Les applications sont de moins en moins monolithiques.  
De plus en plus d'interactions entre les services afin de  
fournir un produit intéressant.

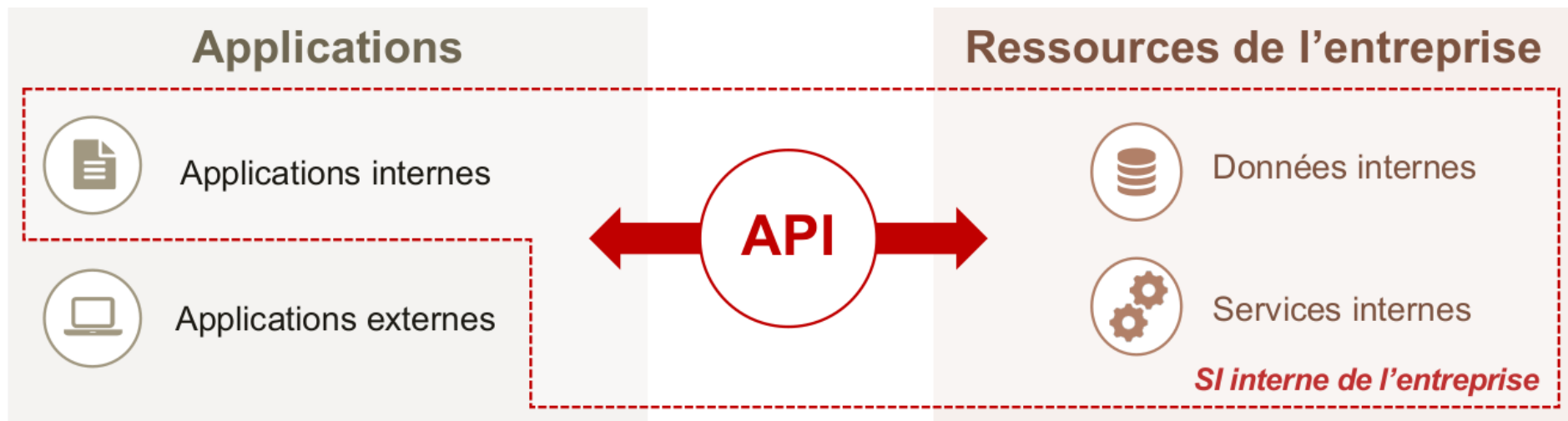
- Mobile Applications
- Single Page Applications
- Microservices
- Serverless
- Open Data
- Obligations légales
- ....



# Positionnement des APIs

L'API , l'une des réponses à cet écosystème multicanal et interactif

L'API est une interface entre les ressources de l'entreprise et des applications internes et externes





# Contraintes des APIs

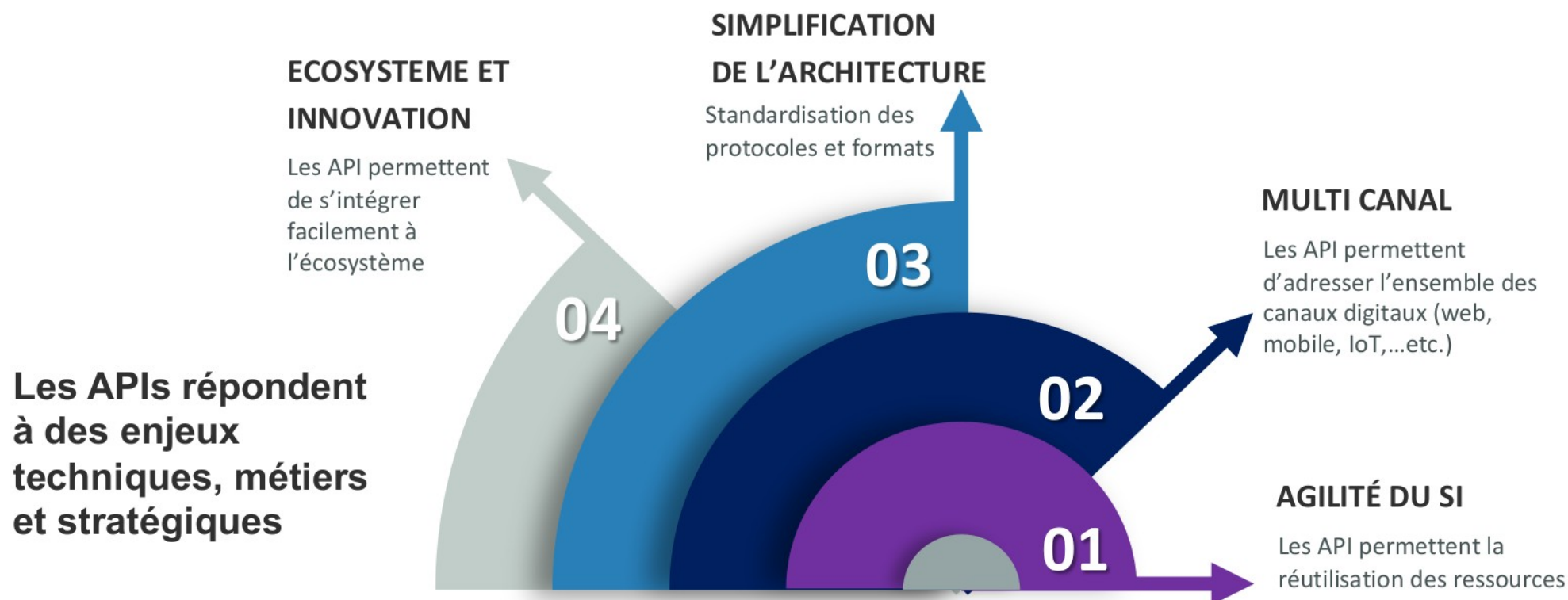
---

En réponse à cette écosystème, les APIs doivent être :

- flexibles, extensibles et réutilisables
- faciles à utiliser et compréhensibles
- conformes à la *Separation of Concerns*
- compatibles avec le plus de technologies possibles
- Le plus légères possible (clients et serveurs)
- Performantes, scalables et sécurisées.



# Enjeux des APIs





# Indicateurs de qualité

---

## Simplicité

Intégration facile

Developer  
Experience  
(parcours simple,  
documentation  
complète,...)

## Disponibilité

Horaires

Continuité du  
service



## Consommation

Nombre d'appels

Indicateur du  
« succès » de l'API

## Scalabilité

Adaptation aux  
variations de  
volume

Maintenance &  
performances







# Introduction

---

Écosystème et enjeux

**Fondations**

SOAP vs REST

Alternatives à SOAP et REST



# Roy Thomas FIELDING et HTTP 1.1

---

**Roy Fielding** a défini REST en 2000 dans sa thèse de doctorat « *Architectural Styles and the Design of Network-based Software Architectures* » à l'université de Californie à Irvine<sup>3</sup>.

Le style d'architecture REST a été développé en parallèle du protocole HTTP 1.1 de 1996 à 1999, basé sur le modèle existant de HTTP 1.0 de 1996.



# Définition et objectifs

---

**REST** (« *REpresentational State Transfer* ») est un ensemble de contraintes pour créer implémenter des services web.

- Le *World Wide Web* est lui-même un système conçu selon l'architecture REST.

L'objectif de REST est l'interopérabilité entre les systèmes sur Internet.

L'utilisation d'un protocole comme HTTP permet aux systèmes REST d'améliorer la séparation des tâches, la généricité, la fiabilité et la réactivité.

*NB : Notez que dans le SI Legacy, l'application du REST n'est pas toujours envisageable. Dans ce cas, il est possible d'utiliser d'autres protocoles comme le SOAP, par exemple.*

# Modèle de maturité des services web

## Niveau 0.

RPC sur HTTP (par ex SOAP)

## Niveau 1.

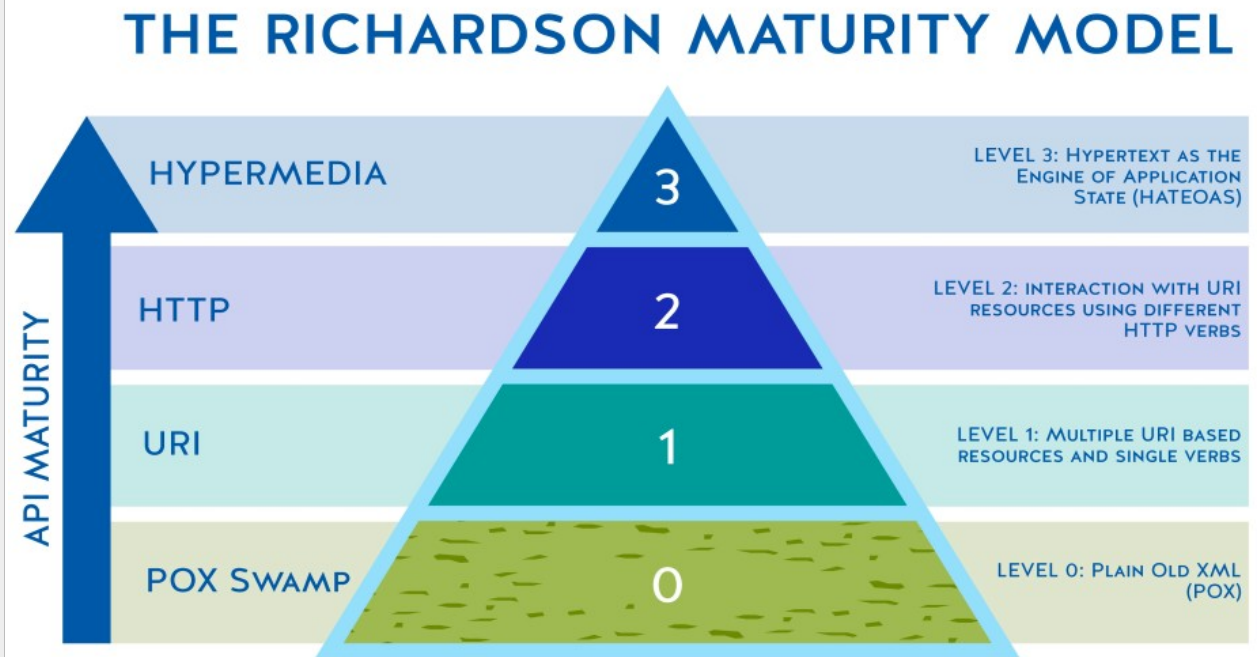
URI identifiant une ressource

## Niveau 2.

Verbes et code retour HTTP identifiant l'intention

## Niveau 3.

HATEOAS, description des ressources disponibles



NORDICAPIS.COM



# Exemple Niveau 0 (XML-RPC)

*Prendre RDV avec un médecin*

---

**POST /appointmentService** HTTP/1.1

[Différentes entêtes]

```
<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

-----

HTTP/1.1 200 OK

[Différentes entêtes]

```
<openSlotList>
```

```
  <slot start = "1400" end = "1450">
```

```
    <doctor id = "mjones"/>
```

```
  </slot>
```

```
  <slot start = "1600" end = "1650">
```

```
    <doctor id = "mjones"/>
```

```
  </slot>
```

```
</openSlotList>
```



# Niveau 1 : Les ressources

---

Un ressource pour un docteur

```
POST /doctors/mjones HTTP/1.1
<openSlotRequest date = "2010-01-04"/>
```

-----

```
HTTP/1.1 200 OK
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

-----

Une ressource pour une plage horaire

```
POST /slots/1234 HTTP/1.1
<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```



# Niveau 2. Verbes et codes retours

---

**GET /doctors/mjones/slots?date=20100104&status=open**

----

HTTP/1.1 200 OK

```
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

-----

**POST /slots/1234 HTTP/1.1**

```
<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

-----


HTTP/1.1 **201 Created**

Location: slots/1234/appointment

```
<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

# Niveau 3.

## Hypertext As The Engine Of Application State



---

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
```

```
----
```

```
HTTP/1.1 200 OK
```

```
<openSlotList>
```

```
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
```

```
    <link rel = "/linkrels/slot/book"
```

```
      uri = "/slots/1234"/>
```

```
  </slot>
```

```
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
```

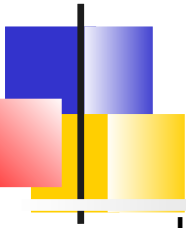
```
    <link rel = "/linkrels/slot/book"
```

```
      uri = "/slots/5678"/>
```

```
  </slot>
```

```
</openSlotList>
```





# Contraintes de l'architecture

Le style d'architecture ReST impose 5 contraintes qui permettent de définir des systèmes hypermedia distribués

- **Separation of Concerns (Client / Server)** : L'API ReST n'est pas concernée par l'affichage, les interactions utilisateur.  
Tous ces éléments doivent être gérés par le client (Ex. : application web frontend).
- **Stateless (sans état)**: Une API ReST ne doit pas maintenir de session ou de contexte.
- **Layered (Organisation en couches)**: La présence de connecteurs intermédiaires doit être transparent pour le client et le serveur (composant de cache / sécurité etc...).
- **Uniforme** : L'interface est uniforme à tous les niveaux (identification de manière unique des ressources, auto-descriptifs,...)
- **Cacheable** : Il doit être possible de mettre les ressources en cache à tous les niveaux
- **Code à la demande** (Optionnel) : Permet au serveur d'étendre la fonctionnalité d'un client en transférant le code exécutable (Peu utilisé dans les faits)



# Introduction

---

Écosystème et enjeux  
Fondations

**SOAP vs REST**

Alternatives à SOAP et REST



# Services Web

---

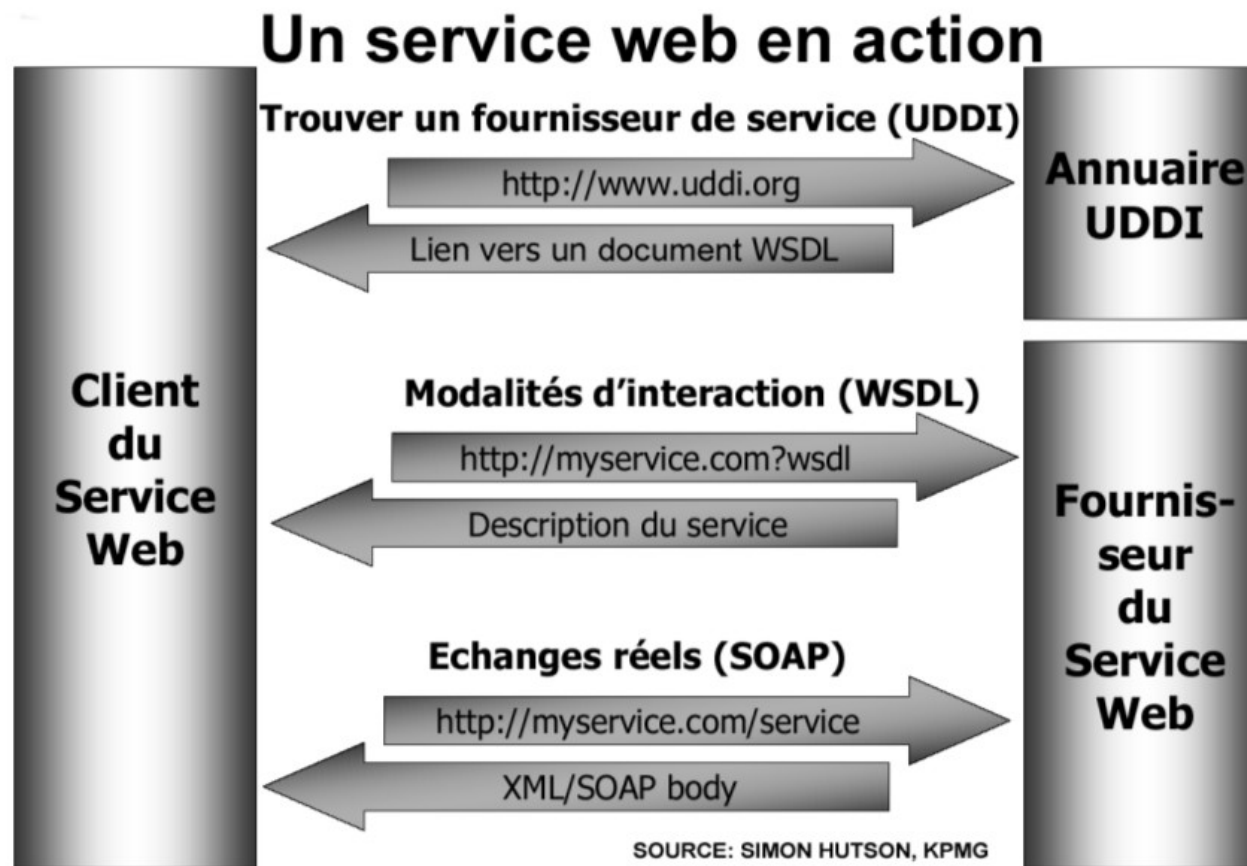
Les services Web destinés à être consommés par d'autres systèmes ont démarré avec SOAP<sup>1</sup> :

- Protocole basé sur XML et les schémas
- Format auto-descriptif : WSDL  
=> Rigueur, verbosité

Actuellement, les services RestFul sont privilégiés :

- Basé sur JSON/YML
- Utilise uniquement HTTP(S)
- Basé sur des conventions
- OpenAPI  
=> Plus léger, plus facile à développer

# Modèle SOAP





# Apports de XML SOAP

---

Les services WEB SOAP bénéficient des fonctionnalités de XML :

- leur grande interopérabilité,
- leur extensibilité
- et leurs descriptions pouvant être traitées automatiquement (WSDL).
- Validation des données échangées via les schémas XML
- Outils connexes *XPath*, *XQuery*, *XLink*



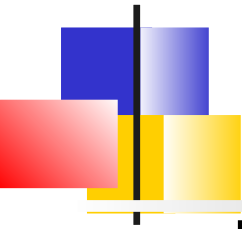
# Inconvénients de XML

---

**Rigueur** : L'utilisation de schéma crée des exigences supplémentaires  
=> Surcoût sur un projet

**Verbosité** : Le volume des données à échanger est fortement augmenté par les balises, les références aux schémas, les espaces de noms.  
=> Les performances sont affectées

**Exigences sur le client** : Le client doit intégrer un parseur XML



# Usage des Services Web

---

Les services Web SOAP sont utilisés principalement pour l'intégration entre systèmes (ESB par exemple).

- Les architectures *SOA (Service Oriented Architecture)* s'appuyant sur SOAP n'ont pas eu le succès escompté.

Les services RestFul se sont répandus :

- Pour l'intégration entre systèmes
- Pour la monétisation de services métiers
- Pour bâtir des SI avec les architectures micro-services



# Introduction

---

Écosystème et enjeux

Fondations

SOAP vs REST

**Alternatives à SOAP et REST**





# Introduction

---

En dehors de SOAP et REST, d'autres protocoles existent et peuvent présenter des alternatives intéressantes :

- **gRPC** (Google 2015) : Appel de méthodes distantes au-dessus d'HTTP. Pas limité aux verbes HTTP, Utilise *Protobuf* pour la sérialisation et la génération de code DTO
- **GraphQL** (Facebook 2012) : Permet au client de définir les données qu'il veut recevoir
- **Netflix Falcor** : Permet de modéliser toutes les données backend (de différents micro-services) sous la forme d'un seul objet Virtual JSON sur un serveur Node



# *gRPC*

*gRPC* est un framework OpenSource de style RPC construit au dessus de ***HTTP/2***

- Permet de définir tout type d'appels de fonction, plutôt que de sélectionner des options prédéfinies telles que PUT et GET dans le cas de REST.

En remplacement de JSON, il utilise ***Protocol Buffers*** pour sérialiser des données structurées.

- Après avoir défini la structure des données, le compilateur de *Protocol Buffer* génère les classes d'accès aux données dans différents langages
- Les données sont ensuite compressées et sérialisées dans un format binaire



# Example

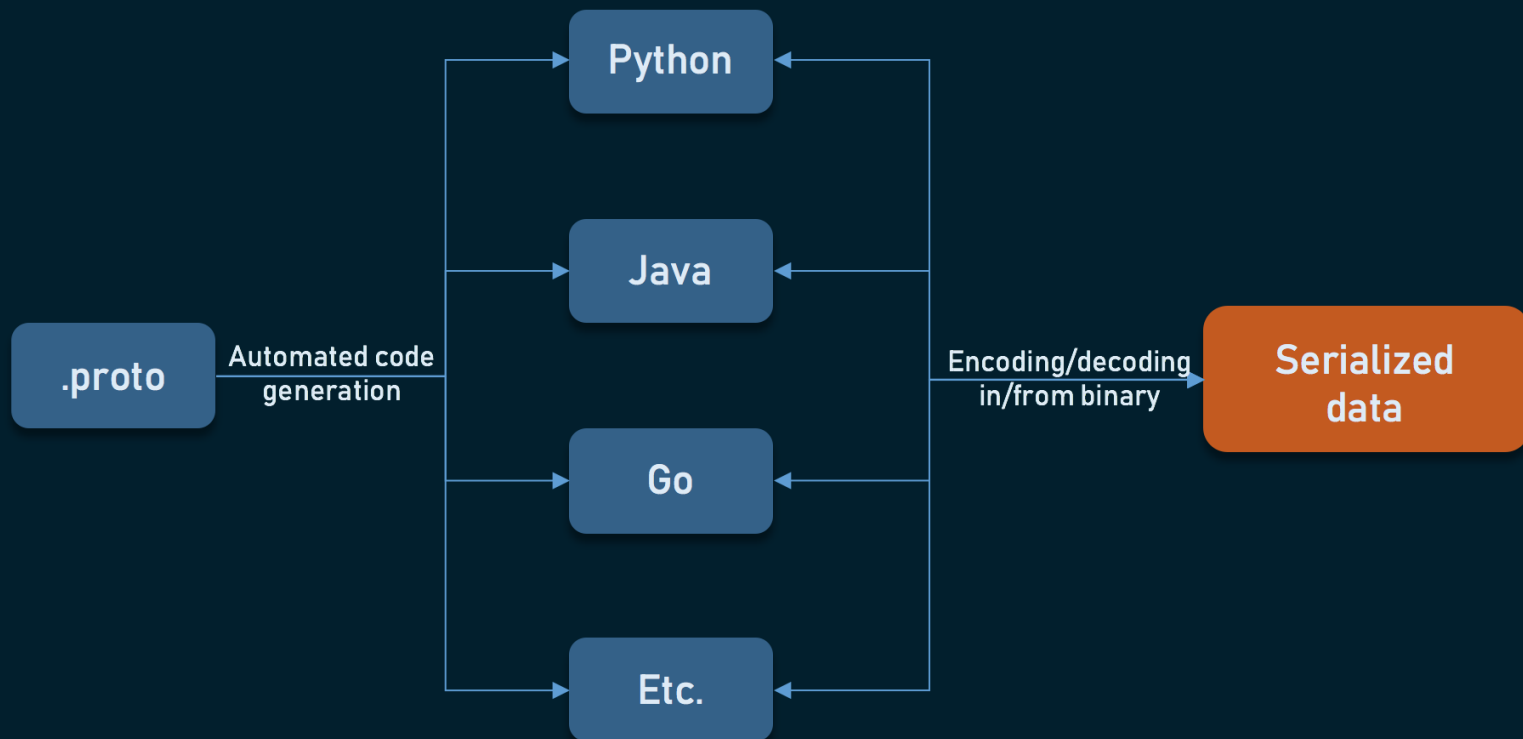
```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

# Protocol Buffers

## DATA SERIALIZATION WITH PROTOCOL BUFFERS





# Avantages de *gRPC*

**Messages légers.** Selon le type d'appel, les messages spécifiques à *gRPC* peuvent être jusqu'à 30 % plus petits que les messages JSON.

**Haute performance.** Selon différentes évaluations, *gRPC* est 5, 7 et même 8 fois plus rapide que la communication REST+JSON.

**Génération de code intégrée.** *gRPC* a automatisé la génération de code dans différents langages de programmation, notamment Java, C++, Python, Go, Dart, Objective-C, Ruby, etc.

**Plus d'options de connexion.** Alors que REST se concentre sur l'architecture requête-réponse, *gRPC* prend en charge le streaming de données avec des architectures pilotées par les événements : streaming côté serveur, streaming côté client et streaming bidirectionnel.



# Faiblesses de *gRPC*

---

**Manque de maturité** : Support minimal pour les développeurs en dehors de Google, peu d'outils pour HTTP/2 et Protocol Buffer.

**Prise en charge limitée des navigateurs** : *gRPC* s'appuie sur HTTP/2, on ne peut pas appeler directement un service gRPC à partir d'un navigateur Web et on doit utiliser un proxy.

**Format non lisible par l'homme** : Les développeurs doivent s'équiper d'outils pour debugger

**Courbe d'apprentissage plus difficile** : Protocol Buffer + HTTP/2



# Introduction

---

## ***GraphQL (Graph Query Language)*** :

Spécification d'un langage de requête et d'un environnement d'exécution

- Dernière release spéc : 2018, Draft en cours
- Développé par Facebook en 2012, puis OpenSource en 2015
- API Web : alternative à REST et SOAP
- Support de Spring en cours de release

C'est le client qui détermine la réponse de l'appel Web plutôt que le serveur



# Exemple

---

Requête POST :

```
{
  orders {
    id
    productsList {
      product {
        name
        price
      }
      quantity
    }
    totalAmount
  }
}
```

Réponse :

```
{
  "data": {
    "orders": [
      {
        "id": 1,
        "productsList": [
          {
            "product": {
              "name": "orange",
              "price": 1.5
            },
            "quantity": 100
          }
        ],
        "totalAmount": 150
      }
    ]
  }
}
```





# Principe

---

Un service *GraphQL* est créé

- En définissant un schéma : Types de données et leurs champs
- En fournissant les fonctions permettant d'accéder à chaque champ

A l'exécution, lors de la réception d'une requête<sup>1</sup>, le service

- Vérifie si la requête correspond au schéma
- Exécute toutes les fonctions nécessaires pour récupérer les données

1. Typiquement via HTTP(S), mais la spéc. ne le précise pas



# Netflix Falcor

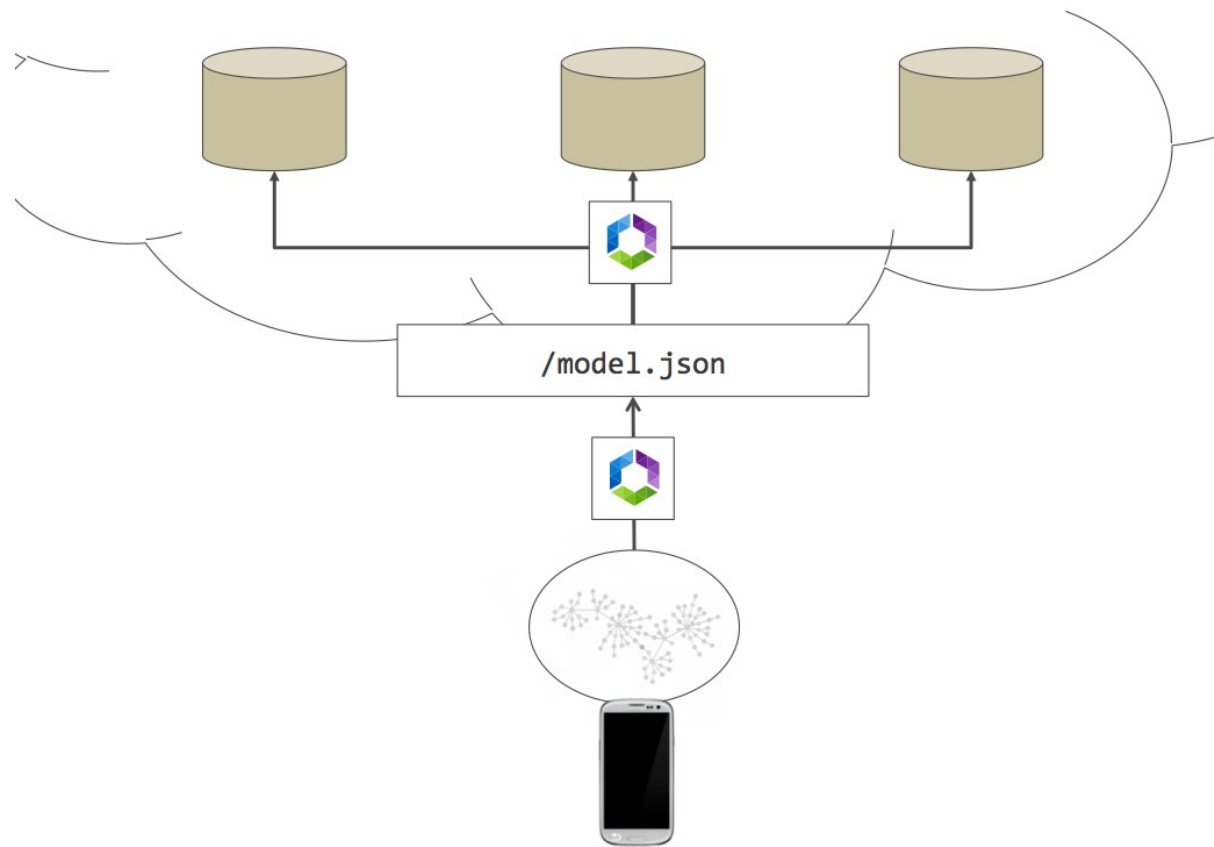
---

***Falcor*** est un middleware.

Il ne remplace pas le serveur d'applications, la base de données ou votre framework MVC.

*Falcor* peut être utilisé pour optimiser la communication entre les couches d'une application nouvelle ou existante.

# Modèle JSON global





# Sous-ensemble

---

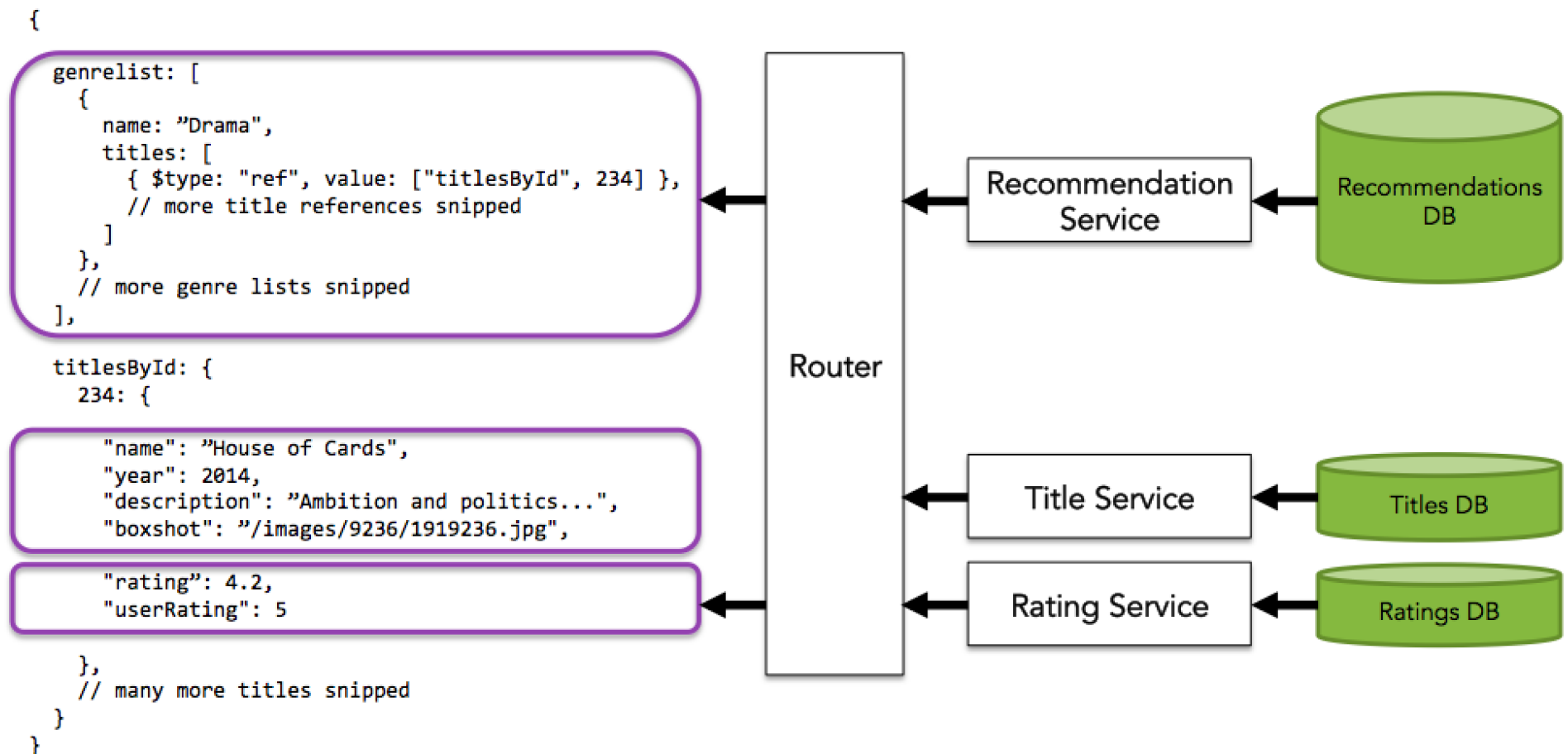
Les clients demandent des sous-ensembles de la ressource JSON globale à la demande.

L'API est tout simplement la donnée

```
/model.json?paths=["user.name", "user.surname", "user.address"]
```

```
GET /model.json?paths=["user.name", "user.surname", "user.address"]
{
  user: {
    name: "Frank",
    surname: "Underwood",
    address: "1600 Pennsylvania Avenue, Washington, DC"
  }
}
```

# Routeur Falcor





# Définition d'API

---

## **Recommandations pragmatiques**

Conventions

Gestion du versionning

Bonnes pratiques de conception

*Open Api Spécification*

*Consumer Driven Contract*



# Objectifs

---

Le but initial est de répondre aux besoins associés à la *User eXperience* et la *Developer eXperience*

Les principaux objectifs des APIs :

- Généricité.
- Facilité d'implémentation
- Extensibilité
- Performance
- Scalability



# Pourquoi sortir une API ?

---

Principalement 3 objectifs dans l'ouverture d'une API

- L'innovation : appropriation des développeurs et son utilisation dans de nouvelles applications
- Revenus : Exploitation directe de votre API dans des services professionnels
- Développement : Utilisation de l'API dans des applications tierces qui vont distribuer le webservice.





# API ouverte/fermée

---

Si l'API permet une réutilisation libre et peu contraignante alors elle sera une API considérée comme **ouverte**.

Si on veut contrôler fortement l'utilisation de votre API, les usages qui vont en être faits, avec par exemple une licence commerciale, la non publication des données collectées, alors votre API sera considérée comme **fermée**.

=> Nécessité de en place des systèmes de sécurité



# Niveaux de sécurité

---

Suivant le type d'utilisateur et les modalités d'utilisation vous choisirez le niveau de sécurité adapté

- **APIkey** pour autoriser la connexion à votre API et le suivi de consommation
- **Quotas** et limitation du nombre de requêtes : pour contrôler l'utilisation de votre API
- Inscription utilisateur : login/mot de passe pour les interactions site-serveur
- **OpenID Connect** , **Oauth 2.0** , pour les services qui nécessitent des interactions serveur-serveur et des ACLs
- **Cryptage SSL** des données , pour les données sensibles. Considérer que toutes les données non cryptées peuvent être interceptées.



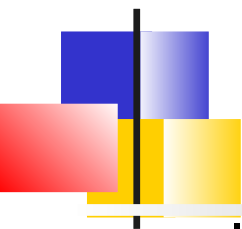
# Agilité

---

Pensez Agile ! N'attendez pas d'avoir une API complète, sortez en *beta*

Si vous avez plusieurs services ou un service complexe, n'attendez pas d'avoir une API qui fait tout avant de la proposer aux développeurs de votre écosystème.

- Proposer quelque chose de simple au début, mais qui fonctionne!
- Utiliser vos early adopters pour créer l'API qui leur convient



# *Developer eXperience*

---

Il est crucial de fournir aux développeurs des API simples à prendre en main avec les outils adaptés.

- Existe un large consensus autour des clients JavaScript. Ne nécessitant pas la maîtrise totale du langage objets.
- Console d'interrogation de l'API, pour voir les réponses en fonction des URL appelées et pour aider à debugger rapidement
- Information sur le statut, la charge, la disponibilité de l'API
- Statistiques d'utilisation personnalisées de votre API afin de manager le trafic
- Fournir des SDK et des librairies dans les langages qu'ils utilisent : Javascript, PHP, .NET, python, Ruby, etc.
- Une bonne documentation didactique et efficace
- Pour chaque appel ou fonctionnalité, fournir un exemple de code et le résultat généré, si possible interactif.



# Suivi et monitoring

---

Suivre les comportements de consommation de votre API

- Utiliser un outil de management de trafic d'API pour identifier qui sont vos utilisateurs
- Analyser les usages et optimiser vos services



# Évolutivité

---

Les API vont changer et évoluer, vous devez anticiper et prévenir votre écosystème de ces changements :

3 conseils qui vous éviteront tout désagrément :

- Développer une API simple au début au design en adéquation avec les attentes des clients , que vous monterez en puissance avec les développeurs
- Faites des versions qui se maintiennent dans le temps
- Publier votre roadmap pour que les développeurs aient une visibilité sur leur engagement.



# Définition d'API

---

Recommandations pragmatiques

## **Conventions**

Gestion du versionning

Bonnes pratiques de conception

*Open Api Specification*

*Consumer Driven Contract*



# Conventions

---

## Convention de Nommage :

- *kebab-case* pour les URLs.
- *snake-case* ou *camelCase* pour les paramètres HTTP et les champs des ressources.
- *kebab-case* pluriel pour les noms des ressources dans les URLs.

Vocabulaire: Utilisez des noms explicites respectant la métaphore du service.

URLs: Les URLs doivent être construites de la façon suivante

```
1 /resources
2 /blogs
3
4 /resources/:resourceId
5 /blogs/:blogId
6
7 /resources/:resourceId/subresources
8 /blogs/:blogId/posts
```





# Base URL

---

Base URL : La base URL est l'URL de la racine de votre API.

Évitez les URLs complexes :

<https://www.ibm.com/index.aspx/lastCompanyWeBought/service/rest>

Préférez :

<https://api.wichtack.com>



# Media-type

---

Le Media Type habituel défini avec l'entête ***Content-Type*** est ***application/json***.

Il est courant de définir un Media Type spécifique pour une API

Exemple : `application/vnd.github+json`.

- Cependant les Media Type *application/vnd\** ne sont pas standards et peuvent éventuellement poser des problèmes avec certaines librairies ou connecteurs (Ex.: Web Application Firewall).

Il est recommandé de rejeter les requêtes ne présentant pas le bon entête *Content-Type*.

Lorsque le client ne présente pas le bon Media Type dans l'entête *Accept*, on peut renvoyer du HTML.



# Propriété *id*

---

Les ressources ont un identifiant unique dans une propriété qui est conventionnellement: *id*.

La propriété *id* doit être uniforme.

Dans le cas d'une ressource immuable, chaque modification peut produire un nouvel *id* faisant référence à une nouvelle ressource.



# Polymorphisme

---

Il peut arriver qu'une ressource de type collection contienne plusieurs ressources de types légèrement différents.

Par exemple, des produits de type différents : livres et films.

Attention :  
le polymorphisme est à utiliser avec parcimonie !

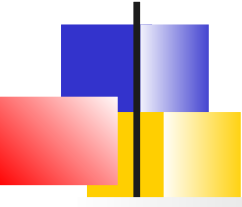
```
{
  "objects": [
    {
      "id": "1",
      "author": {"id": "3"},
      "price": {"amount": 10, "currency": "EUR"},
      "type": "book",
    },
    {
      "duration": 5400,
      "id": "2",
      "price": {"amount": 6, "currency": "USD"},
      "type": "movie"
    }
  ]
}
```



# Règles REST

## Règles REST à respecter pour la création d'API REST

Règle	Description
Separation of concerns (Client / Server)	Ne pas se préoccuper de l'affichage du client
Stateless	Ne pas maintenir de session ou de contexte
Layered	Rendre implicite les connecteurs intermédiaires pour le client et le serveur
Uniforme	Identifier chaque ressource de manière unique et canonique avec l'URI
Cacheable	Permettre de mettre les ressources en cache à tous les niveaux (front, connecteur intermédiaire, back, etc.) et d'utiliser les implémentations standards de cache HTTP



# Format d'échange

Il existe principalement 2 formats de représentation de données pour la spécification des API :

- **JSON** : *JavaScript Object Notation* – Notation Objet issue de JavaScript.  
Le plus utilisé
- **YAML** : *Yet Another Markup Language*



```
1 {
2   "swagger": "2.0",
3   "info": {
4     "version": "v2",
5     "title": "SwaggerDemo API",
6     "description": "Customers API to demo Swagger",
7     "termsOfService": "None",
8     "contact": {
9       "name": "Hinault Romaric",
10      "url": "http://rdonfack.developpez.com/",
11      "email": "hinault@monsite.com"
```

```
1 swagger: '2.0'
2 info:
3   version: v2
4   title: SwaggerDemo API
5   description: Customers API to demo Swagger
6   termsOfService: None
7   contact:
8     name: Hinault Romaric
9     url: http://rdonfack.developpez.com/
10    email: hinault@monsite.com
```



# HTTP : Les méthodes

---

Dans le protocole HTTP, il existe 9 méthodes (appelées aussi commandes ou verbes) pour spécifier une requête afin d'indiquer quelle action exécutée sur la ressource.

En pratique, 7 méthodes sont fréquemment utilisées.

Méthode	Action	Body autorisé
GET	Consulter une ressource	Non
POST	Créer une ressource	Oui
PUT	Mettre à jour/Remplacer une ressource	Oui
DELETE	Supprimer une ressource	Non
HEAD	Vérifier l'existence d'une ressource et récupérer les méta-données	Non
OPTIONS	Déterminer les verbes disponibles	Non
PATCH	Mise à jour partielle	Oui



# Correspondance CRUD

---

Certaines méthodes HTTP ont une correspondance avec les opérations CRUD couramment utilisées pour la persistance des données en bases de données

Opération CRUD	Méthode HTTP	Action
READ	GET	Consulter une ressource
CREATE	POST	Créer une ressource
UPDATE	PUT	Mettre à jour
DELETE	DELETE	Supprimer





# Définition d'API

---

Recommandations pragmatiques

Conventions

**Gestions du versionning**

Bonnes pratiques de conception

*Open Api Specification*

*Consumer Driven Contract*



# Les différentes approches de versioning

---

Les APIs ReST sont conçues pour être utilisées par de multiples clients (mobiles / web / desktop / partenaires / public...)

Elles évoluent souvent à un rythme différent de celui des clients et doivent en général maintenir plusieurs versions simultanément

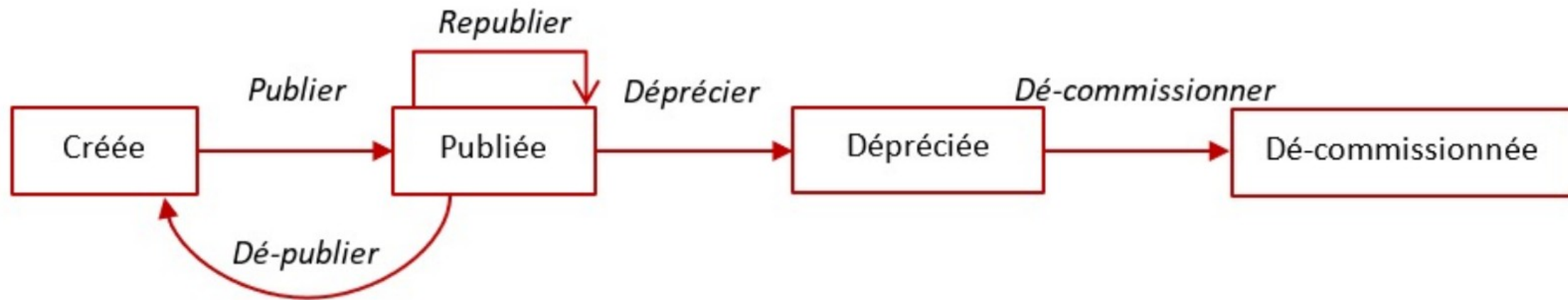
3 principales approches

- **Versioning par Media Type** : Le client indique alors la version de l'API qu'il supporte dans l'entête *Accept*  
Exemple : *Accept: application/vnd.wishtack.v3+json*  
Séduisant mais légèrement complexe à mettre en place.
- **Versioning par URL** : la version est précisée dans le path de l'URL.
- **Pas de versioning** :  
Gestion compliquée des évolutions non-rétrocompatibles



# Cycle de vie des APIs

Le cycle de vie d'une API comprend en général 4 états :



Dans chaque état, l'API est visible pour certains acteurs et invisible pour d'autres .



# Cycle de vie

Etat	Description
Créée	Une API créée n'est pas visible pour les consommateurs d'API sur le portail API. Elle est accessible uniquement aux API Admin pour qu'ils fassent les configurations. Action possible : L'administrateur d'API peut supprimer l'API (elle n'existera plus sur la plateforme) ou la publier
Publiée	Une API publiée est visible dans le portail API. Les utilisateurs peuvent y souscrire et consommer l'API. Action possible : L'administrateur d'API peut mettre à jour l'API (la republier) ou la déprécier
Dépréciée	Une API est dépréciée ou obsolète lorsqu'elle n'a plus de support associé. Une version plus récente a été déployée ou bien l'API n'est plus d'actualité pour l'entreprise. La souscription est interdite mais la consommation est encore possible pour les utilisateurs déjà consommateurs. Action possible : L'administrateur d'API peut uniquement décommissionner l'API (la supprimer de la plateforme)
Décommissionnée	Une API décommissionnée n'est plus visible sur le portail API. Elle n'est plus déployée dans l'environnement de production et n'est plus utilisée/utilisable. Aucune action n'est possible.



# Versionning des API

---

Le versioning des ressources d'API se fait sur 2 digits :  $X.Y$

- **X est le chiffre de version majeure.**  
Incrémenté pour les évolutions non rétrocompatibles seulement. Le consommateur doit s'adapter pour profiter de ces évolutions. Dans le contexte API, la version majeure change rarement.
- **Y est le chiffre de version mineure.**  
Incrémenté pour les évolutions rétrocompatibles de l'interface et/ou des règles de gestion. Ces évolutions sont transparentes pour le consommateur.



# Évolutions non rétrocompatibles et rétrocompatibles

Evolution	Description
non rétrocompatibles	Suppression ou renommage d'une ressource (objet métier) Suppression ou renommage d'une opération Suppression ou renommage d'un attribut ou paramètre (Query, Body et Path param) Ajout d'un attribut ou paramètre obligatoire (Query, Body et Path param) Ajout, suppression ou modification d'une règle métier
rétrocompatibles	Ajout d'une ressource Ajout d'une opération Ajout d'un attribut ou paramètre facultatif (Query, Body et Path param)



# Gestion des versions majeures

---

Les bonnes pratiques pour la gestion des versions majeures des API :

- Seules 2 versions majeures d'une API peuvent être simultanément en production (versions X = nominale et X-1 = dépréciée)
  - Les évolutions sont réalisées sur la version X
  - Les correctifs peuvent être réalisés sur la version X-1 (et reportés sur la version X)
- La montée de version majeure d'une API ne se justifie que dans le cas d'évolutions non rétrocompatibles
- La publication d'une version X entraîne la dépréciation de la version X-1 si elle existe et le dé-commissionnement de la version X-2 si elle existe



# Gestion des versions majeures (2)

---

- Les souscriptions à une version d'API dépréciée sont interdites
- Le calendrier des publications, dépréciations et dé-commissionnements est visible sur le portail API et communiqué par mail aux consommateurs dès que possible.  
Une fonctionnalité d'alerting dans l'API Manager permet de notifier par mail les consommateurs
- Le dé-commissionnement de la version X d'une API peut se faire à l'arrivée de la version X+2 dans un délai déterminé selon certaines contraintes extérieures (source de données, application backend, etc.).





# Gestion des versions mineures et correctifs

---

Dans l'environnement de production :

- Une version majeure X est dans une seule version mineure X.Y
- Une version mineure X.(Y+1) entraîne la suppression de l'ancienne version mineure

Dans l'environnement de qualification :

- Une version majeure X peut être dans 2 versions mineures
- Une version X.Y identique à la version de Production
- Une version X.(Y+1) en cours de développement ou de qualification



# Définition d'API

---

Recommandations pragmatiques

Conventions

Approches de versionning

**Bonnes pratiques de conception**

Open Api Spécification

Consumer Driven Contract



# Granularité de l'API



La granularité d'une API est le niveau de découpage de l'API en un ensemble de ressources :  
il est important de découper « raisonnablement » pour limiter le nombre d'appels.

NB: bonne pratique pour la granularité d'une API est la décomposition par agrégat métier<sup>1</sup>.



# Conception

---

La phase de conception consiste à trouver le juste milieu entre le respect des principes RESTful et une approche.

Très important, une API bien conçue est compréhensible par un développeur sans qu'il ait à lire la documentation

- l'API est auto-descriptive.
- Une API se matérialise directement dans l'URL des requêtes HTTP envoyées au serveur exposant la ressource.

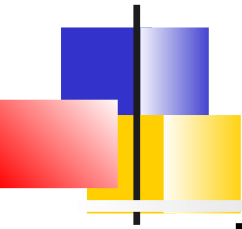
Exemple :

*GET https://api.uber.com/<version>/partners/129/payments*

Intuitivement on comprend que cette ressource concerne les paiement reçus par un partenaire UBER (conducteur indépendant).

# Règles de nommage

Règle	Description
Noms	Utiliser les noms pour décrire la ressource Exemple : POST /payments au lieu de POST /createPayments
Pluriel	Utiliser le pluriel pour gérer les deux types de ressources Collection de ressources ou Instance d'une ressource Exemple Collection : POST /payments Exemple Instance : GET /payments/007
Casse cohérente : <b>snake_case</b> en général <b>Upper Kebab-Case</b> pour les headers	Utiliser le format snake_case car plus lisible en minuscule (fréquemment utilisée par les GAFA). Exemple : short_label et non pas shortLabel Ce format s'applique à tous les champs : L'URL (base path) & l'URI (nom de ressource), Path Param et Query Param, Body de la requête et de la réponse Seule exception, les Headers sont au format Upper Kebab-Case : Exemple : Content-Type
Format court	Utiliser des noms courts Exemple : payments_count et non pas nombre_de_paiements
Label lisible	Accompagner les codes techniques d'un label lisible .Exemple:{ "error_code": "ERR_INT_1ZE23E23", "message":"No message available« }
Pas de caractères spéciaux	Éviter les caractères spéciaux et les espaces dans le nommage des champs
Langues	Utiliser le français ou l'anglais en fonction du contexte de l'utilisateur cible



# Ressource

---

Une ressource représente un objet « entité » décrit par un nom, sur lequel des actions / opérations (CRUD) sont effectuées.

- L'entité représentée par la ressource est généralement une abstraction de nombreuses entités persistantes. (Notion d'agrégat de DDD)
- La conception d'une ressource revient à trouver les bonnes sources des données dans les systèmes d'informations afin de les exposer sous forme de ressources



# Opération d'une ressource

- Une opération est un point de terminaison (ou « endpoint ») HTTP réel qui traite une demande HTTP/S envoyée par une application.
- Une opération est toujours exposée via une ressource. Elle définit une action qui peut être effectuée sur l'entité représentée par la ressource.
- Important : Une opération est autodescriptive grâce à la méthode HTTP qu'elle implémente (GET, POST, PUT, ...).
- Parfois, un chemin explicite est ajouté à la fin de l'URL pour affiner l'action sur la ressource.

Exemple d'une requête avec un chemin implicite :

GET

`https://api.entreprise/calendar/<version>/meetings/{meeting_id}/attendees`

Cette requête renvoie la liste des participants à la réunion.

Exemple d'une requête avec un chemin explicite :

GET

`https://api.entreprise/calendar/<version>/meetings/{meeting_id}/attendees/search`

Cette requête recherche la liste des participants à la réunion.

Le verbe « search » affine l'action si la méthode HTTP seule n'est pas suffisamment claire.



# Paramètres de requête (1)

---

Le protocole HTTP définit 4 types de paramètres qui peuvent être transmis dans une requête :

- ***Path parameter***
- ***Query parameter***
- ***Header***
- ***Body***

Ces paramètres doivent être utilisés selon certaines bonnes pratiques.





# Paramètres de requête (2)

Paramètre	Utilisation	Exemple
<b>Path Parameter</b>	Pour l'identification seulement : <ul style="list-style-type: none"><li>- Uniquement un ID, toujours suivant l'entité à laquelle il se réfère</li><li>- Paramètre obligatoire</li></ul>	http://.../accounts/123/transactions
<b>Query Parameter</b>	Pour la gestion de résultat - filtrer, trier, ordonner, grouper les résultats (paramètres courts) : <ul style="list-style-type: none"><li>- Paramètres techniques optionnels</li><li>- Valeurs sont définies et documentées dans la spécification de l'API</li></ul>	http://.../transactions ? from = NOW & sort = date:desc & limit = 50
<b>Header</b>	Pour la gestion du contexte d'application et de la sécurité <ul style="list-style-type: none"><li>- Utilisé par les navigateurs, les applications clientes et autres pour transmettre des informations sur le contexte de la demande</li><li>- Utilisé pour transmettre les paramètres d'authentification</li></ul>	Authorization : Bearer XXXXXXXXX
<b>Body</b>	Pour les données fonctionnelles <ul style="list-style-type: none"><li>- Utilisé pour transmettre des informations fonctionnelles</li><li>- Doit être un objet JSON</li></ul>	{ "name": "phone", "category": "tech", "max_price": 45 }



# Codes Retours (1)

---

Les codes retours HTTP permettent de déterminer le résultat d'une requête ou d'indiquer une erreur au client.

Ils sont standards et doivent être utilisés d'une façon appropriée pour une gestion efficace des anomalies en production.

5 catégories :

- 100 : Informational
- 200 : Success
- 300 : Redirection
- 400 : Erreur Client
- 500 : Erreur serveur



# Codes 2xx

Code	Description	Cas d'usage
200 – OK	Requête traitée avec succès	Toute requête réussie
201 – Created & Location	Requête traitée avec succès et création d'un document.	Création d'un nouvel objet. Le lien ou l'identifiant de la nouvelle ressource est envoyé dans la réponse
204 – No content	Requête traitée avec succès mais pas d'information à renvoyer.	Mettre à jour ou supprimer un objet (avec une réponse vide)
206 – Partial Content	Une partie seulement de la ressource a été transmise.	Obtenez une liste paginée d'objets



# Codes 3xx

---

Les codes 3xx sont rarement utilisés par les ressources RestFul sauf lors de renommage de l'API

Code	Description
<b>301 – Moved Permanently</b>	Ressource déplacée de façon permanente
<b>302 – Found</b>	Ressource déplacée de façon temporaire



# Codes 4xx (1)

Code	Description	Cas d'usage
<b>400 – Bad Request</b>	La syntaxe de la requête est erronée	<ul style="list-style-type: none"><li>- Format des dates incorrect</li><li>-Envoi de XML au lieu de JSON</li><li>- Integer à la place de string</li><li>- Envoi d'un objet de type non attendu</li><li>- Oubli d'un paramètre obligatoire</li></ul>
<b>401 - Unauthorized</b>	Une authentification est nécessaire pour accéder à la ressource	Jeton d'authentification absent de la Requête ou invalide
<b>403 – Forbidden</b>	Le serveur a compris la requête, mais refuse de l'exécuter : contrairement à l'erreur 401, s'authentifier ne fera aucune différence	Le rôle de l'utilisateur n'est pas suffisant



# Codes 4xx (2)

Code	Description	Cas d'usage
<b>404 – Not Found</b>	Ressource non trouvée	L'objet demandé n'existe pas Exemple : GET/compte/{id} et ce compte n'existe pas.
<b>405 - Method Not Allowed</b>	Méthode connue du serveur mais pas prise en charge pour la ressource	La méthode utilisée n'est pas supportée pour cette URL
<b>406 – Not Acceptable</b>	La ressource demandée n'est pas disponible dans un format qui respecterait les en-têtes "Accept" de la requête	Cela indique qu'il est impossible de servir une réponse qui satisfait aux critères définis dans les en-têtes Accept-Charset et Accept-Language.
<b>409 – Conflict</b>	La demande n'a pas pu être traitée en raison d'un conflit avec l'état actuel de la ressource	Tentative de création d'un nouveau profil utilisateur avec une adresse e-mail déjà existante
<b>429 - Too Many Requests</b>	Le client a émis trop de requêtes dans un délai donné	Throttling



# Codes 5xx

Code	Description	Cas d'usage
<b>501 - Not Implemented</b>	Fonctionnalité réclamée non supportée par le serveur	La méthode (GET, PUT, ...) n'est connue du serveur pour aucune ressource
<b>502 - Bad Gateway ou Proxy Error</b>	Mauvaise réponse envoyée à un serveur intermédiaire par un autre serveur	La réponse du backend n'est pas comprise par l'API Gateway
<b>503 – Service Unavailable</b>	Service temporairement indisponible ou en maintenance	API hors service, en maintenance, ...
<b>504 - Gateway Time-out</b>	Temps d'attente d'une réponse d'un serveur à un serveur intermédiaire écoulé	Timeout dépassé



# Définition d'API

---

Recommandations pragmatiques

Conventions

Approches de versionning

Bonnes pratiques de conception

***Open Api Specification***

*Consumer Driven Contract*





# Introduction

---

La spécification d'une API Rest consiste à définir :

- Les ressources disponibles
- Les méthodes acceptées pour chaque ressource
- Comment fournir les paramètres d'entrée
- Les valeurs de retours :
  - Contenu
  - Statut

Les tests d'acceptation permettront de valider qu'une implémentation respecte la spécification. Il y a un énorme avantage d'automatiser ces tests et de les inclure dans une pipeline de CI/CD



# OpenAPI 3.0

---

Le format de description est dorénavant OpenAPI 3.0 issu du projet Swagger.

- La syntaxe de description est JSON ou YML

**Swagger** est un projet open-source proposant une suite d'outils devenus la référence dans le monde de la conception et la documentation d'API. Il est à l'origine du standard.

**Open API Initiative** est un consortium qui se concentre sur la création, l'évolution et la promotion d'un format de description indépendant du fournisseur.



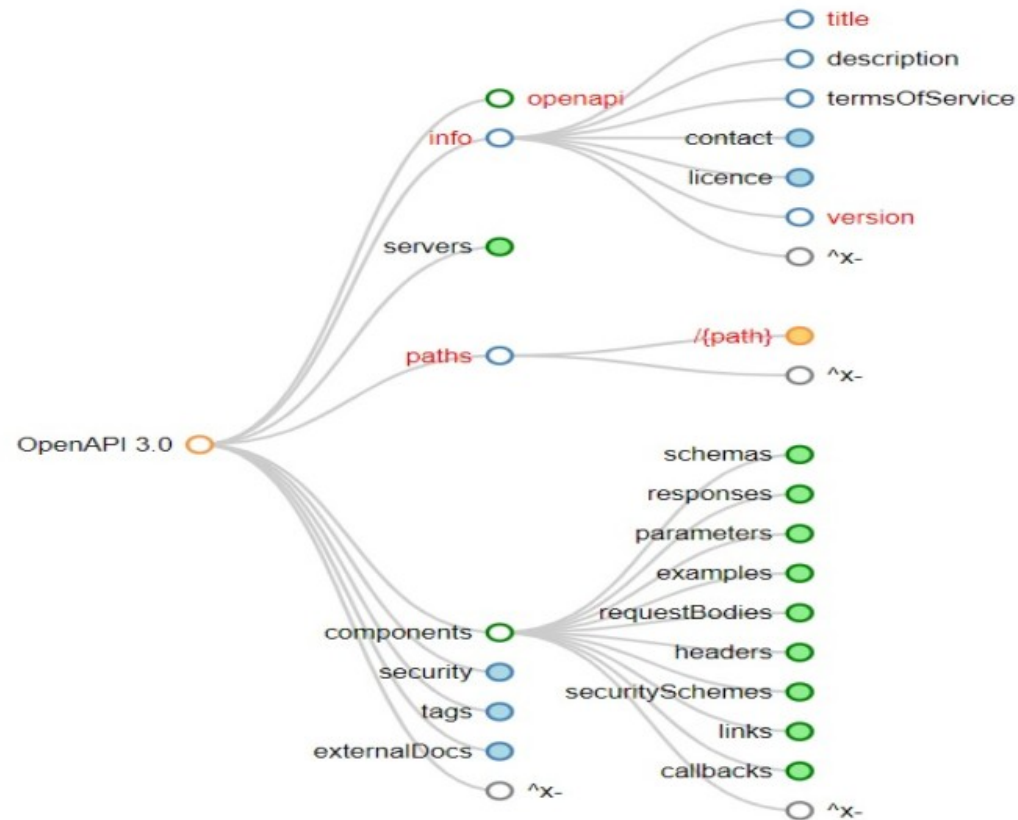
# Usage de OpenAPI 3.0

---

*OpenAPI 3.0* est indispensable dans la mise en place d'APIs RestFul.

- Peut être utilisé en amont pour définir une API et fournir une documentation interactive.
- Peut être utilisé en amont comme générateur de code :
  - Client
  - Mock Serveur
- Peut être généré à partir du code backend pour fournir une documentation interactive

# OpenAPI Map





# Format de description

---

L'*OpenAPI Specification* définit plusieurs objets :

- **openapi** : numéro de version de l'*OpenAPI Specification*
- **info** : Métadonnées sur l'API (description, licence, contact, etc.)
- **servers** : Liste des environnements avec les informations de connexion (URL, etc.)
- **paths** : tous les chemins relatifs aux endpoints et opérations disponibles pour l'API
- **components** : Ensemble d'objets réutilisables pour différents aspects de la spécification (schemas, responses, parameters, headers, etc.)
- **security** : tableau des mécanismes de sécurité qui peuvent être utilisés au travers de l'API pour exécuter les opérations
- **tags** : tableau de tags utilisés pour ajouter des métadonnées
- **externalDocs** : référencement de ressource / documentation externe additionnelle
- **^x-** : balise / préfixe pour ajouter des propriétés personnalisées



# Example

---

```
openapi: 3.0.0
info:
  version: 1.0.0
  title: Sample API
  description: A sample API to illustrate OpenAPI concepts
paths:
  /list:
    get:
      description: Returns a list of stuff
      responses:
        '200':
          description: Successful response
```



# paths

---

Tous les chemins sont relatifs à l'URL du serveur d'API.

- L'URL de requête complète est construite sous la forme `<server-url>/path`.

Ils peuvent utiliser des accolades `{}` pour marquer des parties d'une URL en tant que paramètres de path

Pour chaque chemin, on définit des opérations<sup>1</sup> qui peuvent être utilisées pour accéder à ce chemin.

Les chemins peuvent avoir un bref résumé facultatif et une description plus longue à des fins de documentation

1. OpenAPI 3.0 prend en charge *get, post, put, patch, delete, head, options et trace*.



# Exemple

---

paths:

**/users/{id}:**

summary: Represents a user

description: >

This resource represents an individual user in the system.

Each user is identified by a numeric `id`.

get:

...

patch:

...

delete:

...





# Opérations

---

Une opération comprend les propriétés suivantes :

- **tags** : Pour grouper les opérations
- **summary, description** : Documentation
- **operationId** : Un identifiant
- **parameters** : Liste des paramètres
- **responses** : Liste des réponses possibles
- **externalDocs** : Liens vers de la documentation externe



# Example

---

get:

**tags:**

- Users

**summary:** Gets a user by ID.

**description:** >

A detailed description of the operation.

Use markdown for rich text representation,

**operationId:** getUserById

**parameters:**

- name: id  
in: path  
description: User ID  
required: true  
schema:  
  type: integer  
  format: int64

**responses:**

'200':

description: Successful operation

content:

application/json:

schema:

\$ref: '#/components/schemas/User'

**externalDocs:**

description: Learn more about user operations provided by this API.

url: <http://api.example.com/docs/user-operations/>



# *parameters*

---

Les propriétés d'un paramètre :

- ***name*** : Le nom
- ***in*** : (path, query, header ou cookie)
- ***schema*** :
  - ***type*** : Type Javascript
  - propriétés de validation : int32, enum, minimum, ...
- ***description*** : documentation
- ***required*** : *true, false*



# Exemple

---

parameters:

- in: path

name: id   # Le même nom que dans le path

required: true

schema:

  type: integer

  minimum: 1

description: The user ID



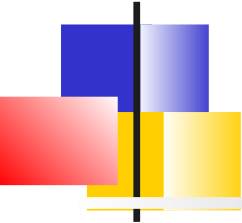
# Corps de requête

---

OpenAPI 3.0 utilise le mot-clé ***requestBody*** pour distinguer le corps de la requête des paramètres

Les propriétés sont :

- ***content*** :
  - Media-types supportés
  - Schéma
- ***description***
- ***required*** (false par défaut).



# *anyOf, oneOf*

---

OpenAPI 3.0 prend en charge ***anyOf*** et ***oneOf***, pour spécifier des schémas alternatifs pour le corps de la requête.

```
requestBody:
  description: A JSON object containing pet information
  content:
    application/json:
      schema:
        oneOf:
          - $ref: '#/components/schemas/Cat'
          - $ref: '#/components/schemas/Dog'
          - $ref: '#/components/schemas/Hamster'
```



# Réponse

---

Chaque opération doit avoir au moins une réponse définie, généralement la réponse réussie.

La propriété ***responses*** est donc un tableau de code statuts, décrits par :

- ***description***
- ***content***
  - ***media-type***
    - ***schéma***



# Exemple

---

responses:

'200':

description: OK

content:

text/plain:

schema:

type: string

example: pong





# Modèle de données (schema)

---

Les types de données sont décrits à l'aide d'un objet ***schema***.

On y trouve les propriétés :

- ***type*** : string, number, integer, boolean, array, object
- En fonction du type, on peut trouver des contraintes sur le format (int32, float), les valeurs (validation, enum, maps)

Les schémas peuvent être combiné par les opérateurs ***oneOf***, ***anyOf***, ***allOf***, ***not***



# Héritage et polymorphisme

---

*OpenAPI* permet de modéliser

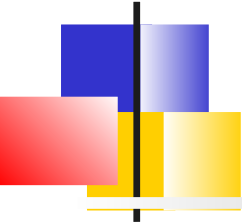
- des relations d'héritage via le mot clé *allOf*
- Du polymorphisme via les mots clé *oneOf* ou *anyOf*



# Exemple héritage

---

```
components:
  schemas:
    BasicErrorModel:
      type: object
      required:
        - message
        - code
      properties:
        message:
          type: string
        code:
          type: integer
          minimum: 100
          maximum: 600
    ExtendedErrorModel:
      allOf:      # Combines the BasicErrorModel and the inline model
        - $ref: '#/components/schemas/BasicErrorModel'
        - type: object
          required:
            - rootCause
          properties:
            rootCause:
              type: string
```



# Exemple polymorphisme

---

```
components:
  responses:
    sampleObjectResponse:
      content:
        application/json:
          schema:
            oneOf:
              - $ref: '#/components/schemas/simpleObject'
              - $ref: '#/components/schemas/complexObject'
```



# Discriminator

---

Pour aider les clients à détecter le type d'objet, on peut ajouter le mot clé ***discriminator/propertyName*** qui pointe vers la propriété qui spécifie le nom du type de données

```
components:
  responses:
    sampleObjectResponse:
      content:
        application/json:
          schema:
            oneOf:
              - $ref: '#/components/schemas/simpleObject'
              - $ref: '#/components/schemas/complexObject'
            discriminator:
              propertyName: objectType
```



# Ajout d'exemples

---

Compléter la spécification par des exemples de paramètres, de propriétés et d'objets a plusieurs avantages :

- Rend la spécification plus claire.
- Permet aux outils de simulation d'API peuvent d'utiliser les exemples pour générer des requêtes fictives.

Cela est effectué via les mots-clés ***example*** ou ***examples***



# Exemple : paramètre

---

parameters:

- in: query

name: limit

schema:

type: integer

maximum: 50

examples: *# Plusieurs exemples*

zero: *# avec des noms distincts*

value: 0 *# une valeur*

summary: A sample limit value

max:

value: 50

summary: A sample limit value



# Exemple requête

---

```
requestBody:
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/User'
      examples:
        Jessica: # Exemple 1
          value:
            id: 10
            name: Jessica Smith
        Ron: # Exemple 2
          value:
            id: 11
            name: Ron Stewart
```





# Exemple réponses

---

responses:

'200':

description: A user object.

content:

application/json:

schema:

\$ref: '#/components/schemas/User'

**examples:**

Jessica:

value:

id: 10

name: Jessica Smith

Ron:

value:

id: 20

name: Ron Stewart



# Définition d'API

---

Recommandations pragmatiques

Conventions

Approches de versionning

Bonnes pratiques de conception

Open Api Spécification

**Consumer Driven Contract**



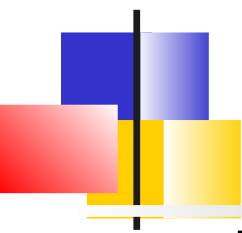
# Consumer Driven Contract Test

***Consumer-driven contract test Pattern***<sup>1</sup> : Vérifier que la « forme » de l'API d'un fournisseur répond aux attentes du consommateur.

Dans le cas REST, le test de contrat vérifie que le fournisseur implémente un point de terminaison qui :

- A la méthode et le chemin HTTP attendus
- Accepte les entêtes attendus
- Accepte un corps de requête, le cas échéant
- Renvoie une réponse avec le code d'état, les entêtes et le corps attendus

1. <http://microservices.io/patterns/testing/service-integration-contract-test.html>



# Spécification par l'exemple

---

Le *Consumer-driven Contract* définit les interactions entre un fournisseur et un consommateur via des **exemples**, i.e les contrats

- Chaque contrat consiste en des exemples de messages échangés durant une interaction

C'est la même idée que l'on trouve dans les approches BDD



# Approches BDD

---

L'approche BDD consiste à spécifier un système via des exemples d'interactions.

Ces exemples sont exécutable et donc automatisables.

L'outil référent dans la matière est *Cucumber*.

L'approche BDD appliquée à une API Rest permet d'aller encore plus loin que OpenAPI 3.0 dans l'automatisation des tests et leur couverture.



# Spécifications par l'exemple, exécutable

---

Les tests d'acceptation sont donc des  
**spécifications exécutables**

- Comme les documents de spécifications traditionnels, ils peuvent être rédigés dans un langage naturel
- La spécification se décrit par des exemples concrets qui illustrent le besoin
- Ils évoluent avec l'avancée du projet et sont versionnés et committés dans le dépôt de sources (SCM)
- Ils sont intégrés dans les pipelines de CI/CD



# Exemple de feature Cucumber

---

**Feature:** Fruit list

To make a great smoothie, I need some fruit

**Scenario:** List fruit

**Given** the system knows about the following fruit:

name	color
banana	yellow
strawberry	red

**When** the client requests GET /fruits

**Then** the response should be JSON:

"""

```
[ {"name": "banana",  
  "color": "yellow"},  
  {"name": "strawberry", "color": "red"}  
]
```

"""



# Code plomberie

---

L'automatisation des tests en langage naturel s'effectue grâce à du code plomberie qui utiliser :

- Soit des apis de bas niveau pour exécuter ses requêtes HTTP et vérifier les réponses
  - *HttpURLConnection* standard.
  - Bibliothèque Apache *HttpClient*.
  - *RestTemplate* Spring
- Soit des librairies spécialisées
  - *RestAssured*
  - *Karate*





# Exemple *RestAssured*

---

```
@Test public void
lotto_resource_returns_200_with_expected_id_and_winners() {

    when().
        get("/lotto/{id}", 5).
    then().
        statusCode(200).
        body("lotto.lottoId", equalTo(5),
            "lotto.winners.winnerId", hasItems(23, 54));

}
```



# Exemple Karate

**Scenario:** create and retrieve a cat

**Given** url 'http://myhost.com/v1/cats'

**And** request { name: 'Billie' }

**When** method **post**

**Then** status 201

**And** match response == { id: '#notnull', name: 'Billie' }

**Given** path **response.id**

**When** method **get**

**Then** status 200

JSON is 'native'  
to the syntax

Intuitive DSL  
for HTTP

Payload  
assertion in  
one line

Second HTTP  
call using  
response data



# Design et outils

---

## **Outils autour OpenAPI 3.0**

Le projet Spring Cloud Contract

API Gateway

API Management : fonctions



# Apports du contrat OpenAPI

---

A partir du contrat, on peut :

- Générer du code backend (Contrôleurs, mapping, modèle), ou client (classe de service et modèle)
- Générer des Mock API : Permettant aux consommateurs de travailler en toute indépendance
- Fournir une documentation interactive
- Éventuellement, permettre des tests d'acceptation (manuels en général)



# Les outils Swagger

---

Principaux outils :

- **Swagger Editor** : Éditeur de spécification OpenAPI
- **Swagger UI** : Visualisation interactive d'une spécification OpenAPI
- **Swagger CodeGen** : Générer des stubs de serveur et des SDK client à partir des spécifications OpenAPI

Mais également :

- Test manuel d'API avec **Swagger Inspector**
- Test fonctionnel, sécurité et performance avec **ReadyAPI**
- Standardisation, Collaboration, Portofolio de projets, Réutilisation avec **SwaggerHub**



# Swagger et Java

---

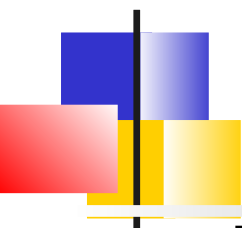
Annotations de description :

- ***io.swagger.core.v3*** : Parfaire la documentation générée à partir du code

Intégration Spring : SpringDoc<sup>1</sup>

- ***springdoc-openapi*** : Génération de la documentation Swagger à partir du code Java
- ***springdoc-openapi-ui*** : Mise à disposition de swagger-ui dans l'application web SpringBoot
- ***springdoc-openapi-maven-plugin*** : Plugin Maven générant la spécification à partir du code

1. Voir : <https://www.baeldung.com/spring-rest-openapi-documentation>



# Fonctionnalités SpringDoc

---

Par défaut,

- La description OpenAPI est disponible à :  
<http://localhost:8080/v3/api-docs/>
- L'interface Swagger à :  
<http://localhost:8080/swagger-ui.html>

SpringDoc prend en compte

- les annotations *javax.validation* positionnées sur les DTOs
- Les Exceptions gérées par les *@ControllerAdvice*
- Les annotations de OpenAPI  
<https://javadoc.io/doc/io.swagger.core.v3/swagger-annotations/latest/index.html>

SpringDoc peut être désactivé via propriété :  
`springdoc.api-docs.enabled=false`



# Exemple annotations

---

```
@Operation(summary = "Get a book by its id")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Found the book",
        content = { @Content(mediaType = "application/json",
            schema = @Schema(implementation = Book.class)) }),
    @ApiResponse(responseCode = "400", description = "Invalid id supplied",
        content = @Content),
    @ApiResponse(responseCode = "404", description = "Book not found",
        content = @Content) })
@GetMapping("/{id}")
public Book findById(@Parameter(description = "id of book to be searched")
    @PathVariable long id) {
    return repository.findById(id).orElseThrow(() -> new
        BookNotFoundException());
}
```





# Quelques autres outils

---

**Stoplight Studio** : outil intuitif pour designer, concevoir et documenter les API  
*<https://stoplight.io/studio/>*

**Apibldr** : outil permettant de définir une spécification OpenAPI via une interface visuelle facile d'utilisation.  
*<https://www.apibldr.com/>*

**OpenAPI GUI** : outil de création et d'édition de définition OpenAPI version 3.0.x  
*<https://github.com/Mermade/openapi-gui>*



# Debug et Testing

---

**SoapUI** : Démarré avec Soap, il s'est adapté aux technologies Rest. Permet la réexécution de tests

**Postman** : est un client ReST très pratique pour analyser, expérimenter, debugger, tester.  
Existe avec extension Chrome.

<https://www.getpostman.com/>

**Insomnia** est une alternative à Postman, pragmatique et light.

<https://insomnia.rest/>



# Générateur de code

---

***open-api-generator*** (fork de swagger-code-gen) : permet la génération de bibliothèques client (nombreux langages), de stubs de serveur, de documentation et de configuration.  
<https://github.com/OpenAPITools/openapi-generator>

***restful-react*** : Pour le framework React.js  
<https://github.com/contiamo/restful-react>

Voir : <https://openapi.tools/#sdk>



# Mocking server

---

**Prism** : Permet de créer une Fake API à partir d'une spécification OpenAPI !

*<https://github.com/stoplightio/prism>*

**openapi-backend** : Configure un nodejs pour mocker une API

*<https://github.com/antiviljami/openapi-backend>*

**GetSandbox**: permet de également générer facilement des bouchons d'API RESTful

*<https://getsandbox.com/>*



# Tests

---

***JSON Generator*** permet de générer facilement des données JSON pour des tests unitaires par exemple

*<https://www.json-generator.com/>*



# Design et outils

---

Outils autour d'OpenAPI 3.0

**Le projet Spring Cloud Contract**

Rôle d'une Gateway

API Management



# Spring Cloud Contract

---

***Spring Cloud Contract*** est un projet qui permet d'adopter une approche *Consumer Driven Contract*

A partir d'une spécification d'interaction entre un producteur/serveur et consommateur/client, cela permet

- De générer des tests d'acceptation côté producteur
- De créer des mocks serveur pour le client



# DSL

---

*Spring Cloud Contract* et son projet principal **Verifier** fournit un **DSL** (*yaml, Groovy, Kotlin ou Java*) qui permet de spécifier le contrat

Le contrat permet alors de produire :

- Les tests d'acceptations complets côté serveurs (*Spock* ou *JUnit*)
- Les définitions de stub JSON à utiliser par *WireMock* lors des tests d'intégration sur le code client.  
Le code de test doit toujours être écrit à la main mais les données de test sont produites.

SCC permet également les tests de routing de message lors de l'utilisation de Spring Cloud Stream





# Exemple Groovy

---

```
import org.springframework.cloud.contract.spec.Contract
Contract.make {
    description "should return even when number input is even"
    request{
        method GET()
        url("/validate/prime-number") {
            queryParameters {
                parameter("number", "2")
            }
        }
    }
    response {
        body("Even")
        status 200
    }
}
```



# Application à Rest

---

Dans le cas de REST, Spring Cloud Contract vérifie que,

- pour une requête qui correspond aux critères de la partie requête du contrat,
- le serveur fournit une réponse conforme à la partie réponse du contrat



# Comparaison entre les syntaxe

---

## YAML

- Plus facile
- S'apparente à OpenAPI mais est différente
- Des tentatives de convertisseur  
<https://github.com/springframeworkguru/spring-cloud-contract-oa3>

## Groovy

- Premier langage a être supporté
- Données dynamiques
- DSL adapté

## Java, Kotlin

- Idem Groovy sans aspect DSL et donc syntaxe plus lourde
- Plus répandu



# Contrats pour http

---

Les éléments haut niveau d'un contrat HTTP sont :

- ***request***: Obligatoire
- ***response*** : Obligatoire
- ***priority***: Optionnel

```
priority: 8  
request:  
...  
response:  
...
```



# Requête

---

Seulement ***method*** et ***urlPath*** sont spécifiés dans la propriété request

Optionnellement, on peut spécifier :

- ***queryParameters***
- ***headers***
- ***cookies***
- ***body***



# Exemple

---

```
request:
  method: PUT
  urlPath: /foo
  queryParameters:
    a: b
    b: c
  headers:
    foo: bar
    fooReq: baz
  body:
    foo: bar
```



# Réponse

---

Obligatoirement un code retour : ***status***

Optionnellement :

- ***headers***
- ***cookies***
- ***body***



# Propriétés dynamiques

---

Le contrat peut contenir certaines propriétés dynamiques : horodatages, identifiants, etc.

Avec Groovy les données dynamiques peuvent être fournies de 2 façons :

- Directement dans le corps
- Dans une section distincte appelée ***bodyMatchers***

Avec YAML, les données dynamiques sont fournies par la section ***matchers***





# Exemple yaml

---

- path: \$.thing1  
type: by\_regex  
value: regexp  
regexType: as\_string
- path: \$.thing2  
type: by\_regex  
predefined: only\_alpha\_unicode



# Tests générés côté producteur

---

```
public class ContractVerifierTest extends BaseTestClass {

    @Test
    public void validate_shouldReturnEvenWhenRequestParamIsEven() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .queryParams("number", "2")
            .get("/validate/prime-number");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);

        // and:
        String responseBody = response.getBody().asString();
        assertThat(responseBody).isEqualTo("Even");
    }
}
```



# Plugin Maven

---

Spring Cloud Contract met à disposition un plugin Maven qui offre 2 principaux objectifs :

- ***generateTests*** : Génère les classes de test validant la spécification
- ***generateStubs*** : Génère un artefact contenant la spécification ... qui pourra servir pour générer le MockServer

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <version>2.1.1.RELEASE</version>
  <extensions>true</extensions>
  <configuration>
    <baseClassForTests> org.formation.BaseTestClass</baseClassForTests>
  </configuration>
</plugin>
```



# Tests Consommateur

---

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
@AutoConfigureJsonTesters
@AutoConfigureStubRunner(
    stubsMode = StubRunnerProperties.StubsMode.LOCAL,
    ids = "com.baeldung.spring.cloud:spring-cloud-contract-producer::stubs:8090")
public class BasicMathControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void given_WhenPassEvenNumberInQueryParam_ThenReturnEven()
        throws Exception {

        mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=2")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().string("Even"));
    }
}
```



# Design et outils

---

Outils autour d'OpenAPI 3.0  
Le projet Spring Cloud Contract  
**Rôle d'une Gateway**  
API Management



# Introduction

---

Une API Restful a potentiellement de nombreux clients différents (Application Mobile, Web, Applications de partenaires, ...) qui nécessitent différentes données.

Il est donc difficile d'offrir une API unique qui conviennent à tous les clients



# Inconvénients des appels directs

---

Les clients peuvent également besoin d'invoquer différentes APIs (architecture micro-services)

L'appel directe des différents services par les clients a de nombreux inconvénients :

- Peut obliger les clients à faire plusieurs requêtes pour récupérer les données dont ils ont besoin
- Le manque d'encapsulation. Les clients doivent connaître chaque service et leur API.
- Les services peuvent utiliser des mécanismes IPC qui ne sont pas pratiques pour des clients à l'extérieur du firewall



# API Gateway Pattern

---

**API gateway Pattern**<sup>1</sup> : Implémente un service qui est le point d'entrée de l'application micro-service pour les clients externes

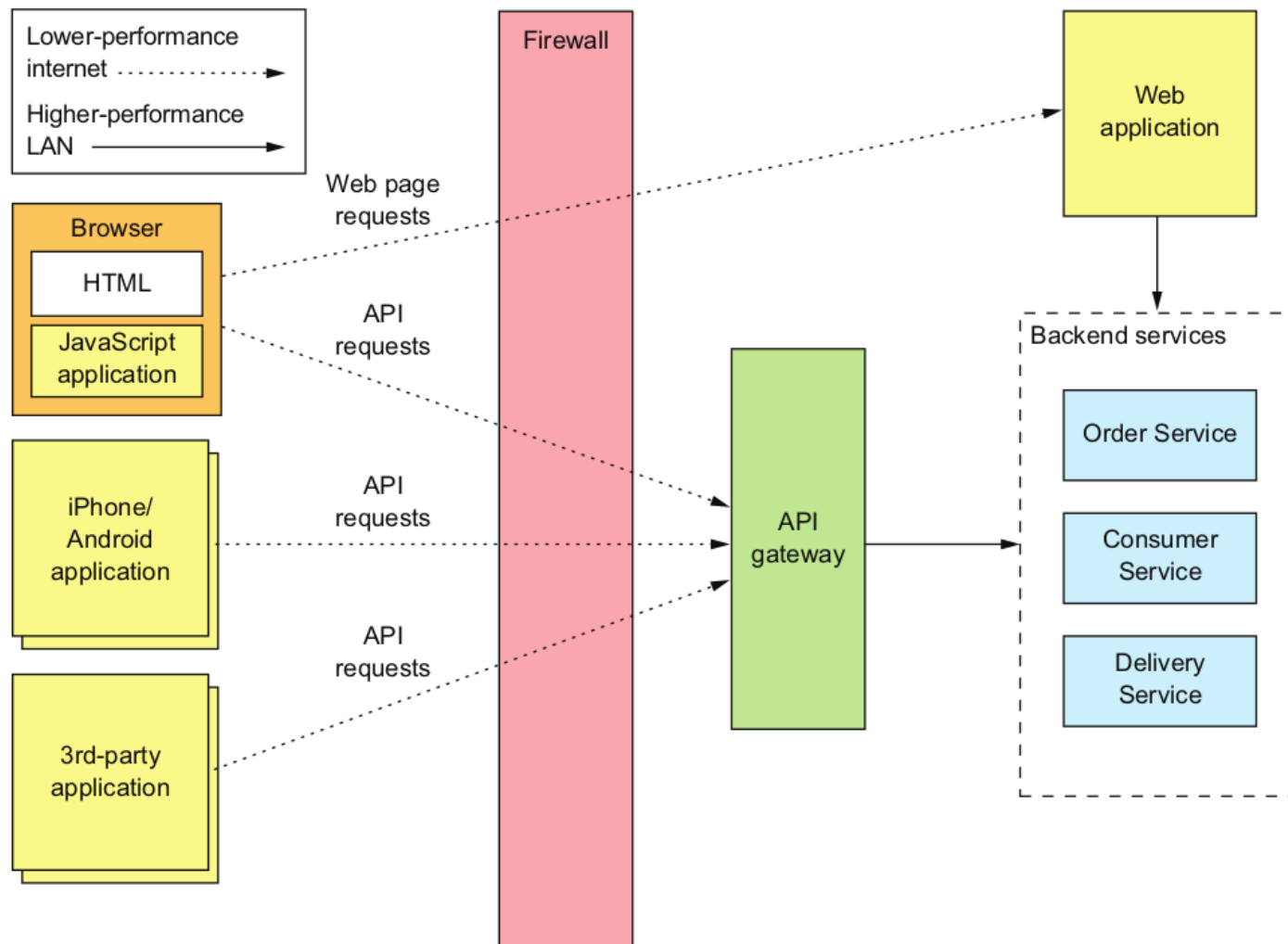
Le service API Gateway est alors responsable du routage des requêtes, de la composition d'API, de la traduction de protocole, de l'authentification et d'autres fonctions

Similaire au pattern *Facade* en Objet

1. <http://microservices.io/patterns/apigateway.html>



# API Gateway





# Fonctions de la Gateway

---

Routage : En fonction d'une table de routage, la gateway transfère les requêtes aux services backend. La table de routage peut s'appuyer sur tous les composants HTTP (URL, entêtes, paramètres de requêtes)

Composition d'API : Plusieurs appels vers les services backend sont alors agrégés

Traduction de protocoles : Traduction de requêtes GraphQL en requêtes REST par exemple

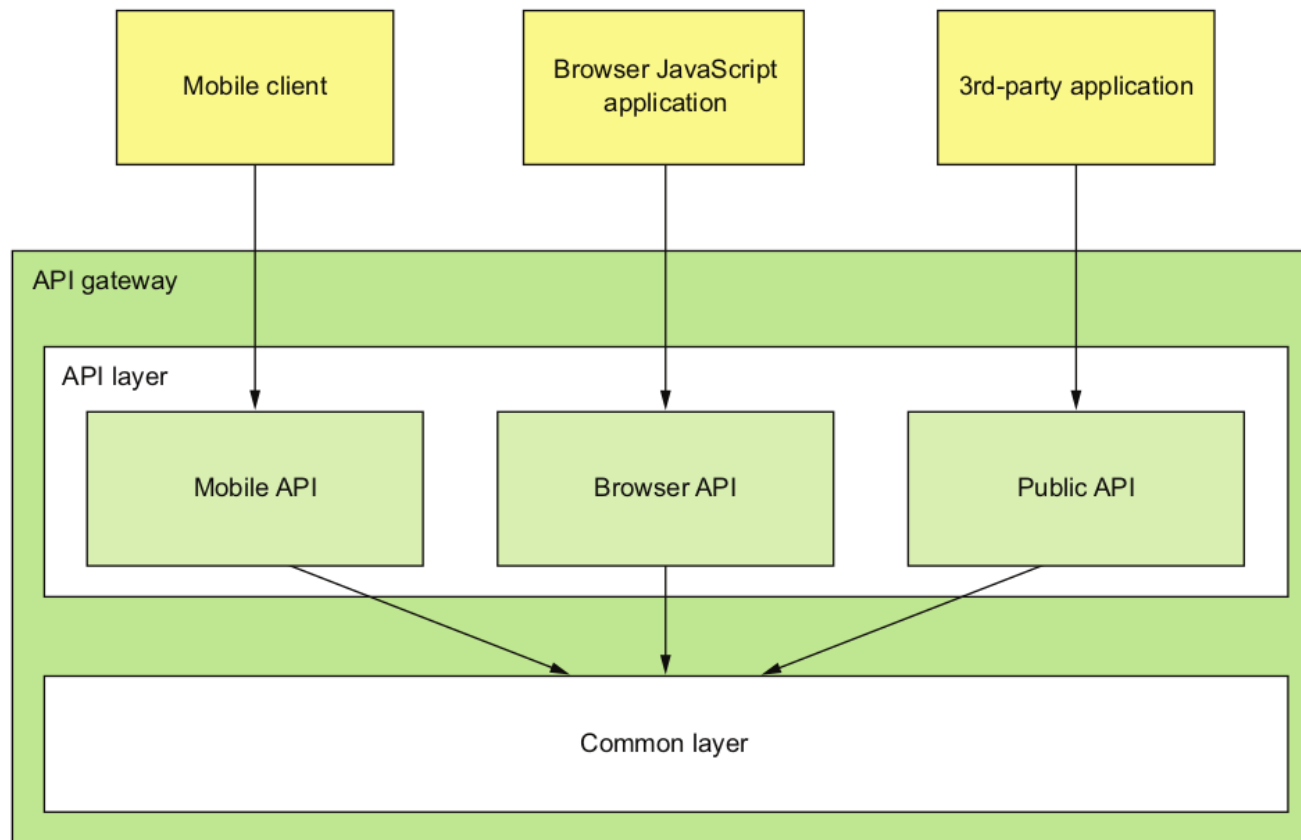
API spécifique par clients : La gateway n'offre pas la même API pour un mobile que pour les partenaires externes

Fonctions transverses (edge functions)<sup>1</sup> : Implémentation de l'authentification, de l'autorisation, du cache, du log de requêtes, ....

1. On peut également implémenter ces fonctions dans un service dédié en amont de la gateway

# Design de la Gateway

Chaque opération de l'API est :  
Soit un simple routage  
Soit une composition





# Organisation

---

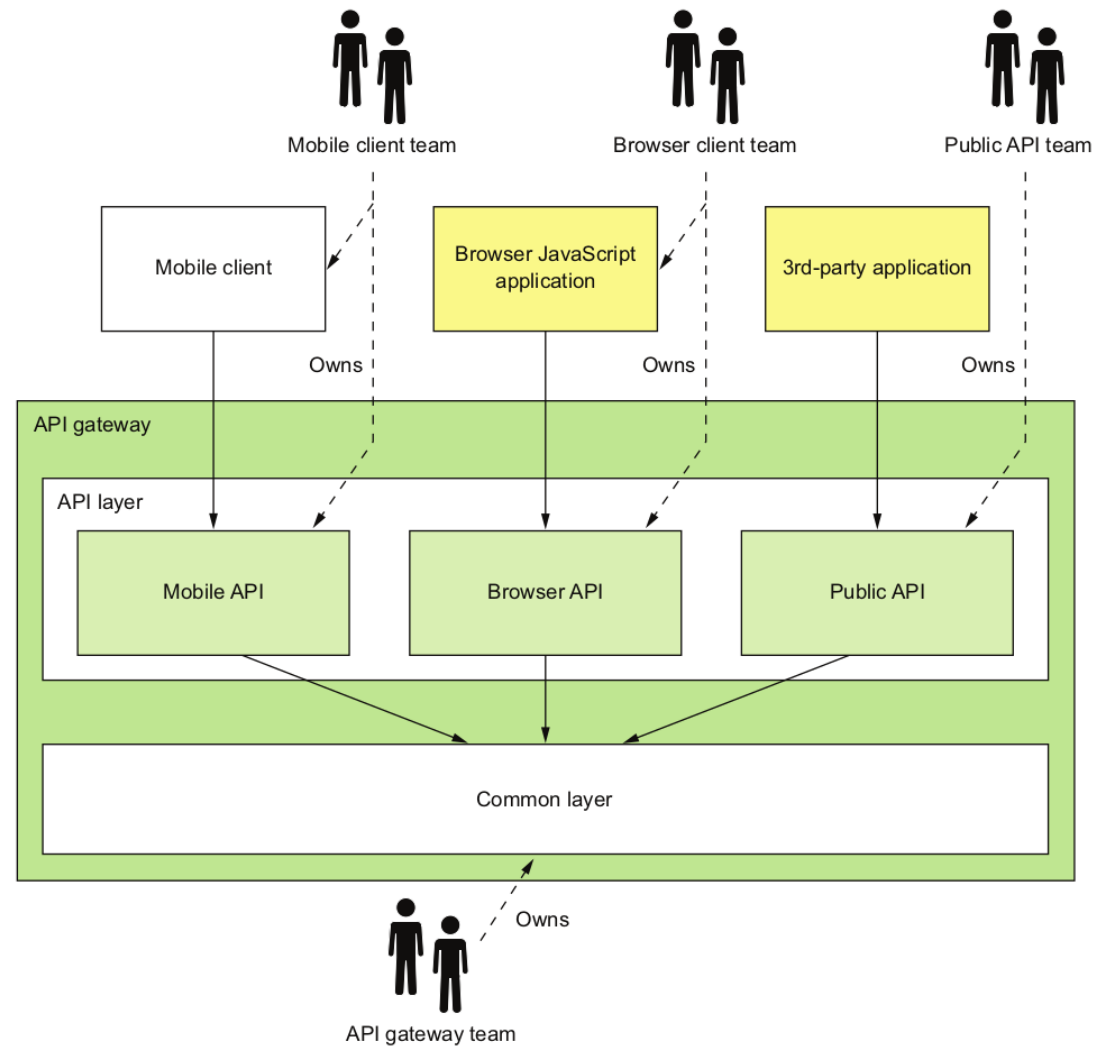
Qui est responsable du développement et de l'exploitation de la gateway ?

Une approche efficace, promue par Netflix, consiste de confier le module API d'un client à l'équipe responsable du code client (les équipes d'API mobiles, Web Javascript, ...).

Une équipe Gateway est responsable du module Commun et de l'exploitation de la gateway.

Lorsqu'une équipe cliente doit modifier son API, elle committe ses modifications dans le référentiel source de la Gateway et une pipeline de déploiement continue valide la cohérence.

# Organisation





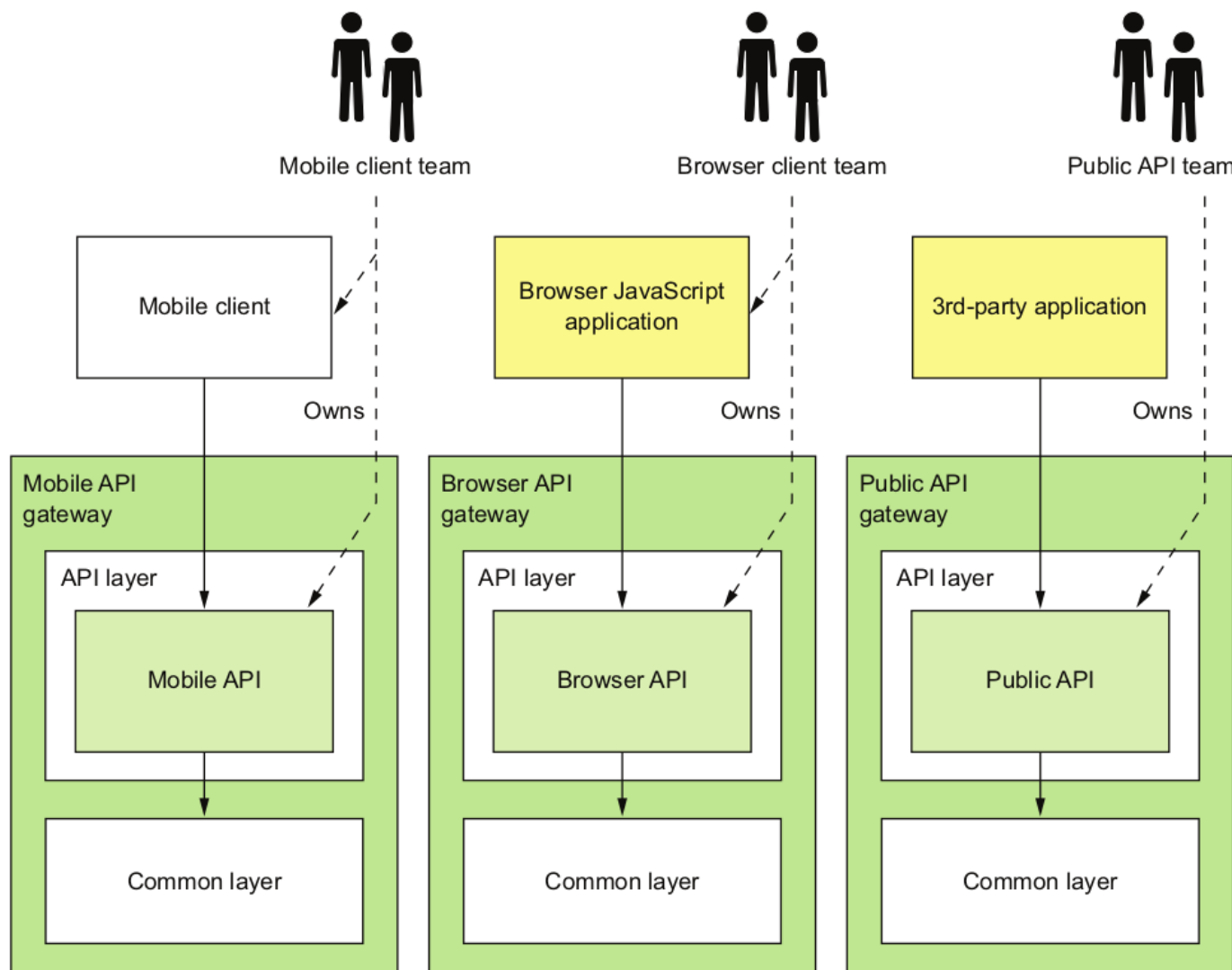
# Backends for frontends Pattern

---

L'organisation précédente est contraire aux principes des micro-services.  
Chaque équipe doit être indépendante.

=> **Backends for frontends Pattern**<sup>1</sup> :  
Implémenter une gateway différente pour chaque type de client.

1. <http://microservices.io/patterns/apigateway.html>





# Alternatives à REST

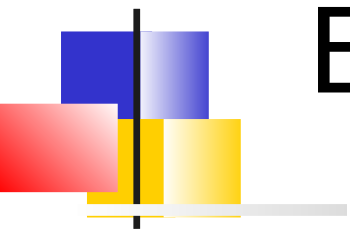
---

Un autre alternative afin que chaque client puisse avoir sa propre API est d'utiliser des technologies comme *GraphQL* ou *Netflix Falcor*.

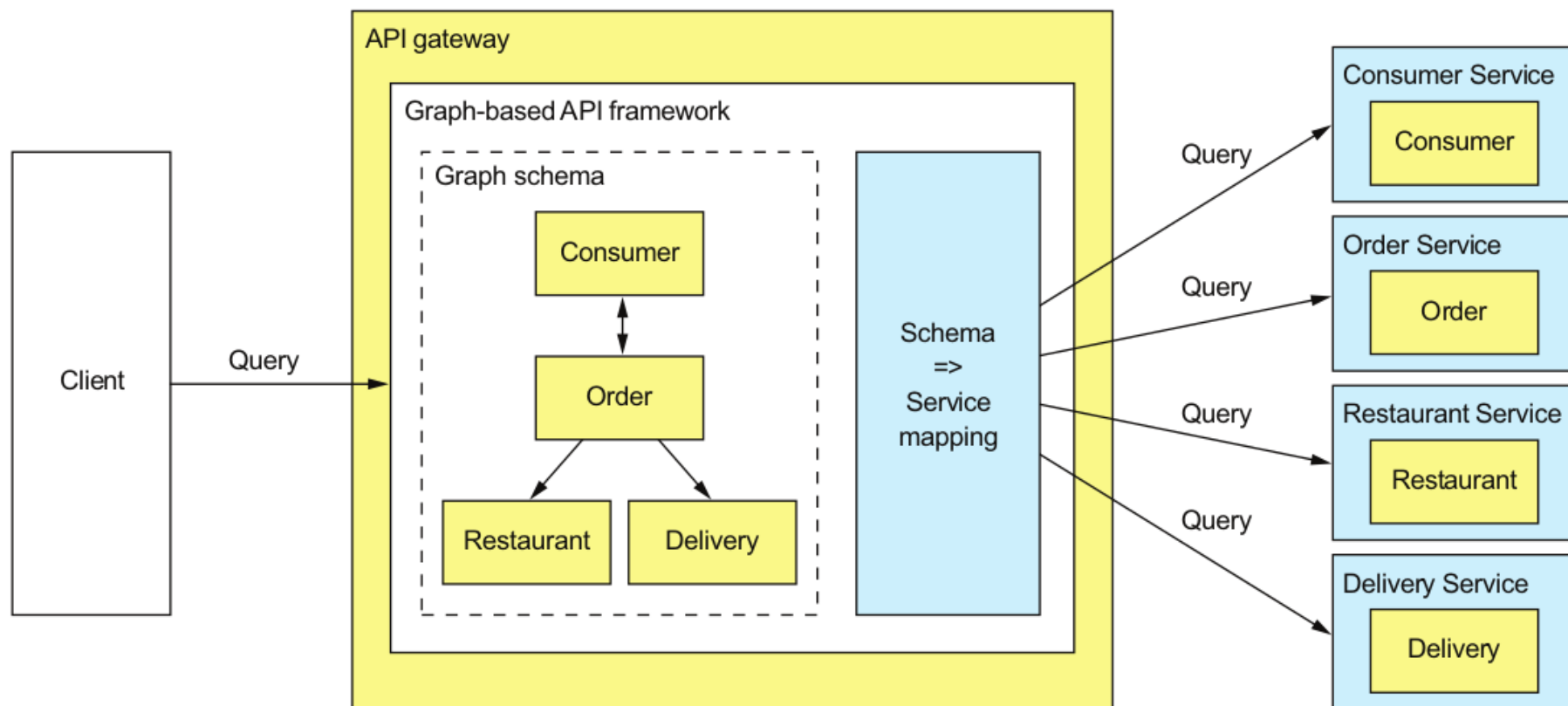
- Ces technologies sont des alternatives au modèle REST.

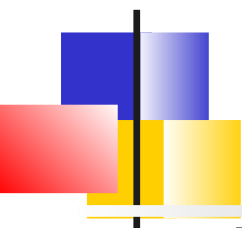
Le client peut demander le graphe de données qu'il veut obtenir dans sa requête





# Exemple GraphQL Gateway





# Bénéfices / Inconvénients

---

## Bénéfices

Encapsule la structure interne de l'application

Réduit les aller-retours entre le client et l'application

## Inconvénients

Un autre service hautement disponible à exploiter

Risque de goulot d'étranglement

Une mise à jour des APIs backend nécessite une mise à jour des APIs de la gateway



# Design issues

---

Performance et scalabilité : Modèle réactif plus approprié

Composition API : Effectuer si possible les requêtes en parallèle. Encore une fois modèle réactif

Traiter les dysfonctionnements : Répliquer la gateway, appliquer le circuit breaker pattern

Doit respecter les mêmes principes que les services backend : Se baser sur un service de discovery et offrir des points d'observabilité



# Design et outils

---

Outils autour d'OpenAPI 3.0  
Le projet Spring Cloud Contract  
Rôle d'une Gateway  
**API Management**

# Fonctionnalités de l'API Manager

API Gateway	Authen/author des accès	Vérification des tokens de sécurité et de l'autorisation d'accès à la maille API	Équilibrage de charge	Répartition de la charge entre les nœuds d'une même API
	Mise en cache	Mise en cache d'une partie des résultats des appels aux API (standards HTTP)	Routage	Découverte des services et routage des échanges
	Médiation et interopérabilité	Médiation protocolaire et forçage de certains standards d'interopérabilité	Metering	Mesure de la consommation de chaque application consommatrice (par API, par jour, ...)
	Alerting	Sur atteinte des quotas ou évènement de sécurité	Throtling	Limitation de la consommation pour une application (ex: 10k appels /jour pour API A)
API Store	Catalogue et documentation	Catalogue d'API éligibles et documentation (compatible avec le format Swagger/OpenAPI)	Souscription au service	Gestion de son compte développeur et souscription aux API sous réserve d'éligibilité
	Sandbox	Outils de test des API (bouchons) par les développeurs d'applications	Suivi de la consommation	Portail de suivi des souscription, de la consommation et de la facturation
API Publisher	Gestion de quotas	Attribution de quotas sur la consommation des API (globaux et spécifiques aux comptes)	Gestion de la visibilité	Quel développeur d'application peut accéder à quelles API ?
	Cycle de vie	Gestion du cycle de vie des API (définition, test, publication, mise à jour, décommissionnement)	Gestion des droits	Gestion de droits d'accès des utilisateurs / systèmes : contrôle sur les API store et Gateway
	Niveaux de service	Différents niveaux de service pour les différentes API	Facturation	Mécanisme de facturation de l'utilisation des API (nombre d'appels, SLA souscrit ...)



# Solutions Historiques

Généralement issues de solutions ESB, réactualisées et transformée pour la gestion d'API REST. On y trouve des solutions très complètes mais souvent assez lourdes et complexes. Les éditeurs fournissent un niveau de support élevé en lien avec leur taille.



Axway est une solution d'API management déployée chez de nombreux clients en France (Total, iCDC, Fortis, BPCE). Elle a prouvé sa robustesse et sa performance dans des cas d'usages réels. Elle fournit des fonctionnalités sécurité avancées (couplage à un anti virus, lutte contre les injections de code, etc.).



La solution de CA est largement reconnue. Elle a été retenue notamment par la BNPP (ITG). C'est une solution complète, qui propose de nombreuses fonctionnalités de médiation et qui assure également une protection contre les failles du TOP 10 OWASP. Néanmoins son administration (via un client lourd) est datée et complexe.



WSO2 est la solution opensource d'APIM leader. Elle est mise en prod et/ou reconnue comme solution cible chez plusieurs de nos clients (Crédit Agricole, Société Générale IBFS, ...). Elle bénéficie également d'une communauté active. A noter qu'une nouvelle version entièrement révisée est prévue pour début 2018.



IBM était en clairement en retard sur le marché, plombé par la gamme Datapower qui a toujours rencontré un succès commercial mitigé. Cette année, sa solution API connect a entièrement été refondue et semble prometteuse. Elle a notamment été adoptées par PSA. La solution est cloud hybride, et supporte les websocket.



Software AG est un leader historique des middlewares SOA avec son produit phare : WebMethods (ESB). Leur solution d'API Management actuelle est clairement en retrait et peu mature par rapport aux autres éditeurs. Cependant, leur produit est en cours de refonte complète.



TIBCO est un leader sur le marché des produits middleware. Ils ont racheté et intégré à leur offre le produit Mashery, leader et pure player du marché, initialement détenu par INTEL. Nous n'avons pas de REX client sur ce produit mais beaucoup de REX positifs sur l'EAI/ESB de TIBCO et les bus de messages(PMU, AG2RLM, etc.).

# Solutions récentes

Plus récentes, et créées directement pour la gestion d'API. Ces solutions, majoritairement open-source et quelquefois exclusivement cloud-native sont parfois plus limitées en fonctionnalités mais plus légères et moins coûteuses.



Solution publiée en 2010 et rachetée en 2016 par Google, Apigee Edge est un API Manager cloud et on-premise récent mais à la politique commerciale très agressive sur le marché européen. La solution est à l'étude chez Malakoff Médéric.



3scale est une Solution initialement full cloud et récemment enrichie d'une offre partiellement on-premise lors de son rachat par Red Hat



APIMan est une solution full on-premise de RedHat. Celle-ci va certainement être fusionnée avec 3scale dans un futur proche.



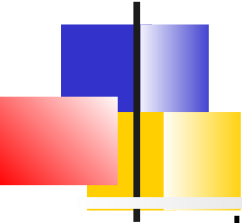
Anciennement Mashape, Kong a été écrit en 2009 en C/Lua et est basé sur Nginx. L'API Gateway fonctionne sur un principe de plugin pour implémenter ses fonctionnalités. Les fonctionnalités d'API Store et de monitoring sont fournies par les solutions (cloud, payantes) Gelato et Galileo. La solution est étudiée par Nexity.



Tyk est un API Manager léger et relativement récent (2014) écrit en Go. Peu de référence à date sur cette solution.



La solution d'API Management de Mulesoft est historiquement une solution purement cloud, qui s'est récemment enrichie d'une édition "on-premise". Celle-ci est complexe à mettre en place et moins aboutie que la version cloud. Par ailleurs, nous n'avons pas connaissance de références significative sur la sphère Banque & Assurance.



# Gestion du patrimoine : Portail des API

---

Le portail des API est un site web pour regrouper et gérer les API comme des produits et rationaliser le processus d'habilitation des développeurs à utiliser les API de l'entreprise.

L'objectif du portail des API est de permettre aux clients ou développeurs :

- D'obtenir des identifiants et gérer leurs profils
- De déclarer leurs applications
- De naviguer dans le catalogue API pour découvrir et tester les API
- D'explorer la documentation des API
- D'obtenir des informations sur le cycle de vie des API

NB : Quand le portail API est exposé à l'extérieur, il devient la vitrine des produits API que l'entreprise propose. Il est important qu'il soit disponible 24h/24h, que la navigation soit intuitive pour les développeurs pour une adhésion forte





# Classification des APIs

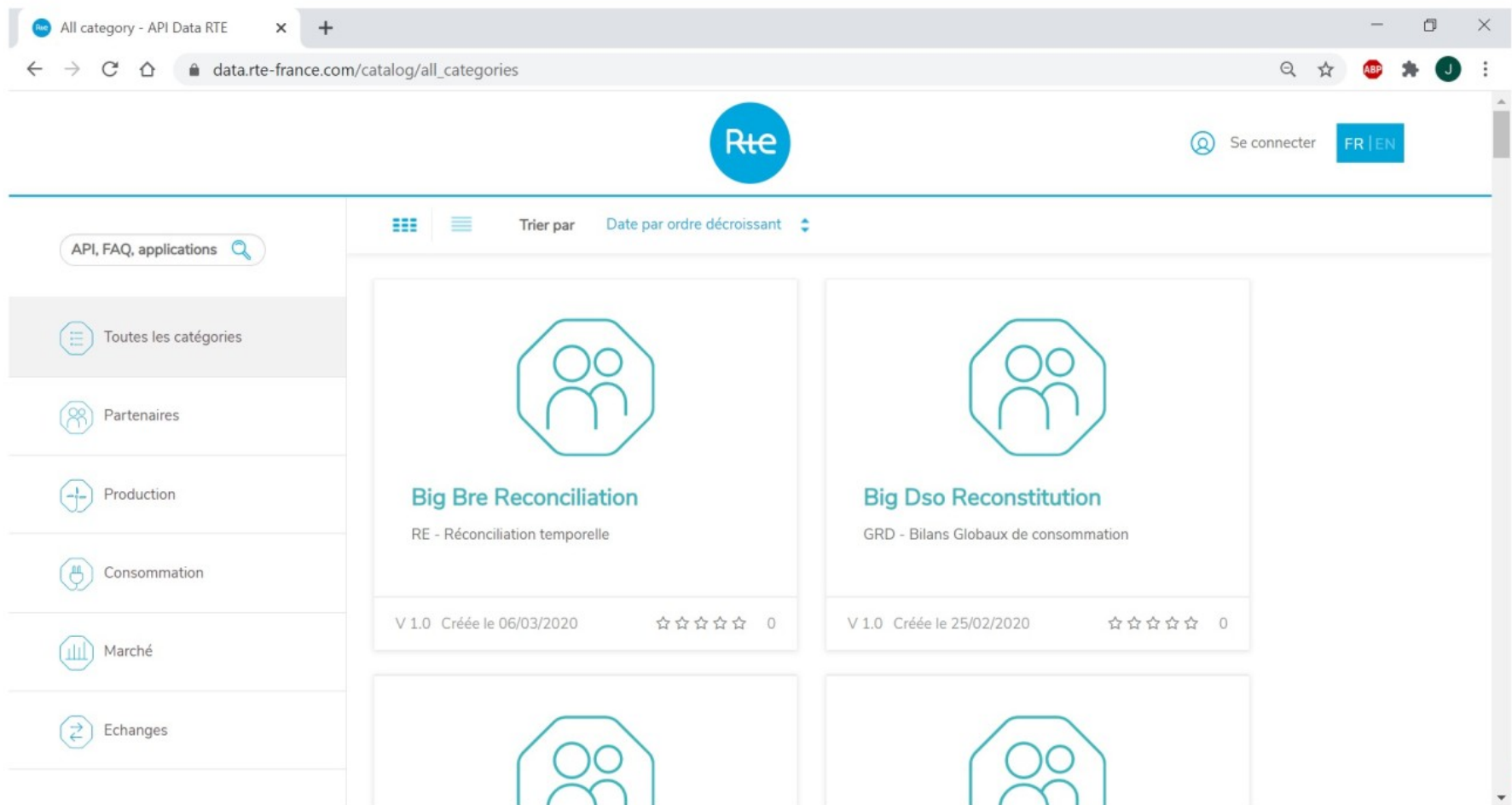
---

La classification des API s'effectue généralement selon les usages, 3 grandes catégories:

- **API Privée** : une API Privée peut être consommée uniquement par applications internes à l'entreprise
- **API Partenaire** : une API Partenaire peut être consommée uniquement par des applications d'entreprises partenaires : dans un grand groupe, une entreprise partenaire peut être une autre filiale du groupe par exemple
- **API Publique** (ou « Open API ») : une API Publique est consommée par toutes les applications, internes ou externes à l'entreprise

La classification des API permet d'afficher ou non les API dans un référentiel ou catalogue API en fonction du profil utilisateur.

# Exemple RTE





# Documentation des API

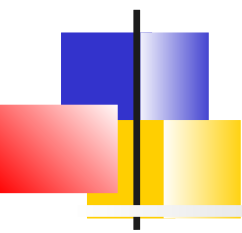
---

Pour chaque API, la documentation contient idéalement :

- La version de l'API
- Une description synthétique de l'objectif fonctionnel de l'API
- La liste de toutes les ressources
- Le contrat d'interface pour chaque ressource et opération :
  - URL de la requête HTTP
  - Requête en entrée, paramètres des requêtes en entrée (path param, query param, ...), champs obligatoires, ...
  - Requête en sortie, codes retours, champs en sortie, ...
- Optionnel : les évolutions à venir dans la prochaine version de l'API

On peut bien sûr s'appuyer sur Swagger afin d'avoir une documentation cohérente avec l'API et constamment à jour

Cette documentation est mise en ligne sur le portail API.



# Implémentation avec la POO

---

## **Architecture logicielle**

Patterns pour la logique métier

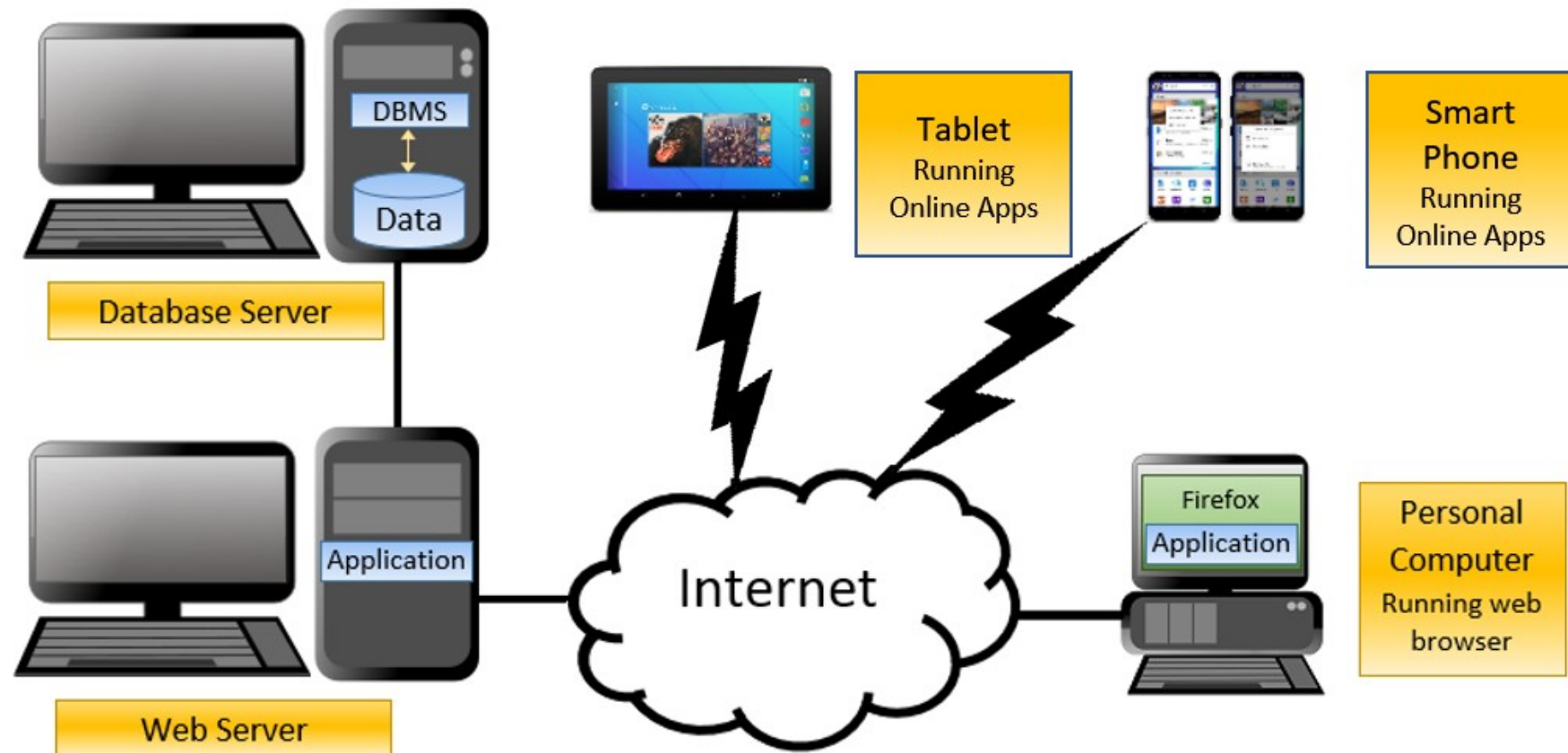
Couche Controller

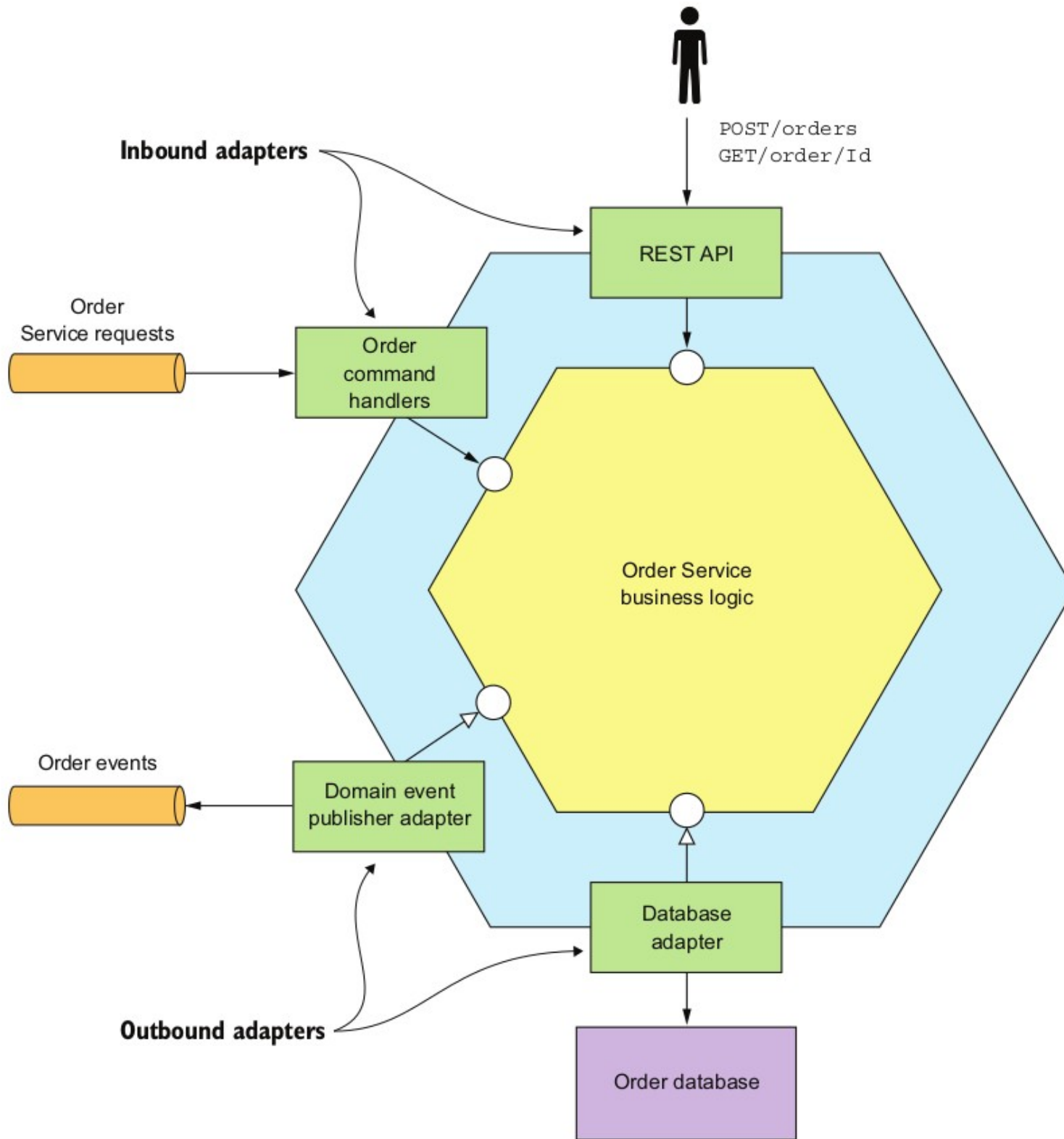
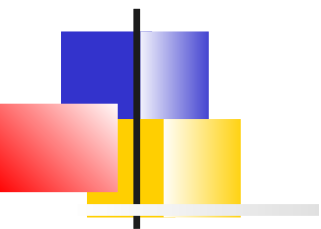
Les problématiques de  
sérialisation/désérialisation

Impacts sur la couche DAO

Exceptions, CORS

# Architecture 3-tiers

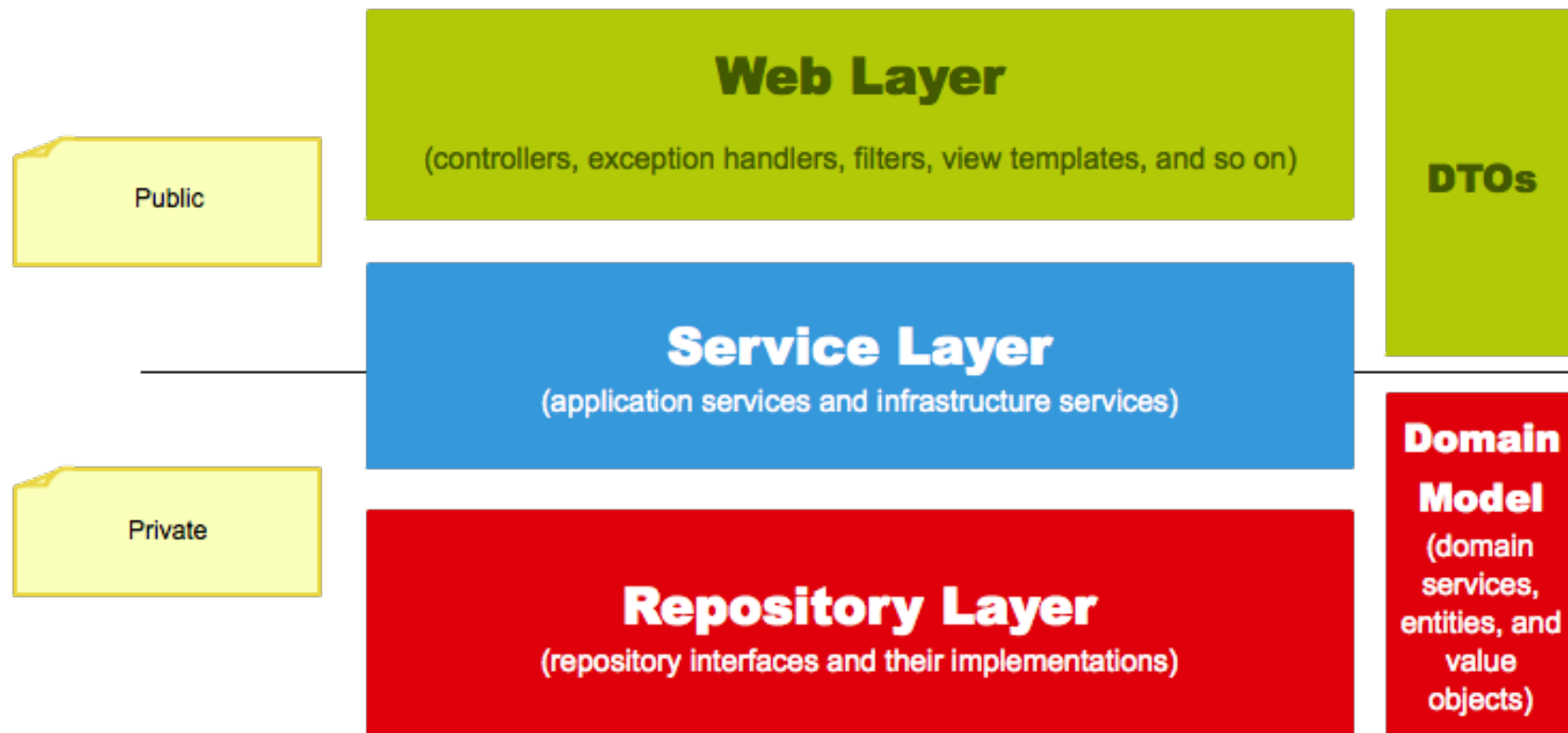






# Couches logicielles

---





# Classes du modèles

---

Elles modélisent les données persistantes du service

- s'appuie généralement sur un framework d'Object Mapping qui convertit les opérations CRUD sur les objets dans la syntaxe appropriée au support de persistance (SQL, NoSQL, ...)  
Exemple : Hibernate, Spring ORM
- Définit les associations entre les entités, leur stratégie de chargement et de sauvegarde
- Peut également, incorporer de la logique métier.  
(Voir *Domain Model Pattern*)





# Exemple : Entités JPA

**@Entity**

```
public class Theme {  
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String label;  
    @OneToMany(cascade = CascadeType.ALL)  
    private Set<MotClef> motclefs = new HashSet<MotClef>();  
}
```

**@Entity**

```
public class MotClef {  
    @Id  
    private Long id;  
    private String mot;  
  
    public MotClef(){}  
}
```



# DAO/Repository

---

Elles implémentent les opérations CRUD sur les objets du modèle ainsi que les opérations de requêtage

Dans le cas d'une BD relationnelle :

- Via JDBC
- Via JPA
- Via une couche supérieure d'abstraction.  
Exemple Spring DATA JPA ou Spring JDBC



# Exemple JPARepository

---

```
public interface MemberRepository
    extends JpaRepository<Member, Long> {

    /**
     * Tous les membres ayant un email particulier.
     * @param email
     * @return
     */
    public List<Member> findByEmail(String email);

    /**
     * Chargement de la jointure one2Many.
     * @param id
     * @return
     */
    @Query("from Member m left join fetch m.documents where m.id =:id")
    public Member fullLoad(Long id);
}
```



# Classes services

---

Elles agrègent plusieurs opérations de persistance en un opération métier

- Généralement transactionnelle (ACID)
- Peut s'appuyer sur d'autres services REST.  
Architecture micro-service
- Implémentent directement la logique métier (Transaction Script Pattern) ou s'appuie sur la logique métier implémenté dans les classes du modèle (Domain Model Pattern)
- Les opérations peuvent être sécurisés



# Exemple Spring

---

**@Service**

```
class UserManagementImpl implements UserManagement {  
  
    private final UserRepository userRepository;  
    private final RoleRepository roleRepository;  
  
    public UserManagementImpl(UserRepository userRepository,  
        RoleRepository roleRepository) {  
        this.userRepository = userRepository;  
        this.roleRepository = roleRepository;  
    }  
}
```

**@Transactional**

**@RolesAllowed("MANAGER")**

```
public void addRoleToAllUsers(String roleName) {  
  
    Role role = roleRepository.findByName(roleName);  
  
    for (User user : userRepository.findAll()) {  
        user.addRole(role);  
        userRepository.save(user);  
    }  
}
```



# DTO

---

Les classes services renvoient généralement des classes modèles ou des classes DTO (Data Transfer Object)

- Les classes DTO sont éventuellement créées pour s'adapter au format JSON de l'API
- Des bibliothèques de mapping peuvent être utilisées pour convertir les objets du modèles en classe DTO

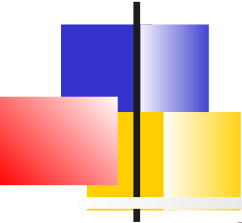


# Couche Web

---

La couche Web est implémentée par des classes contrôleur.

- Les méthodes des contrôleurs sont mappés vers les requêtes HTTP correspondantes
- Elles s'appuient sur la couche service ou directement sur la couche Repository
- Leur responsabilité est de générer la réponse HTTP en cohérence avec les principes RestFul (Code retour, entête, etc..)



# Types des valeurs de retours des méthodes

---

Les types des valeurs de retour possibles pour un contrôleur REST sont :

- Une classe **Modèle ou DTO** qui sera converti en JSON via la librairie Jackson.  
Le code retour est alors 200
- Un objet ***ResponseEntity<T>*** permettant de positionner les codes retour et les entêtes HTTP voulues





# Exemple RestController Spring

---

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @GetMapping(value="/{user}")
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @GetMapping(value="/{user}/customers")
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @DeleteMapping(value="/{user}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long user) {
        // ...
    }

    @PostMapping
    public ResponseEntity<Member> register(@Valid @RequestBody Member member) {

        member = memberRepository.save(member);

        return new ResponseEntity<>(member,HttpStatus.CREATED);
    }
}
```



# Implémentation avec la POO

---

Architecture logicielle

**Patterns pour la logique métier**

Couche Controller

Les problématiques de  
sérialisation/désérialisation

Impacts sur la couche DAO

Exceptions, CORS



# Introduction

---

La logique métier est généralement la partie la plus complexe du service.

L'organisation des classes doit être la plus appropriée à l'application

Même avec une technologie objet, il y a 2 principaux patterns pour organiser la logique métier d'un service :

- Procédurale : *Transaction script pattern*,
- Orienté objet : *Domain model pattern*.

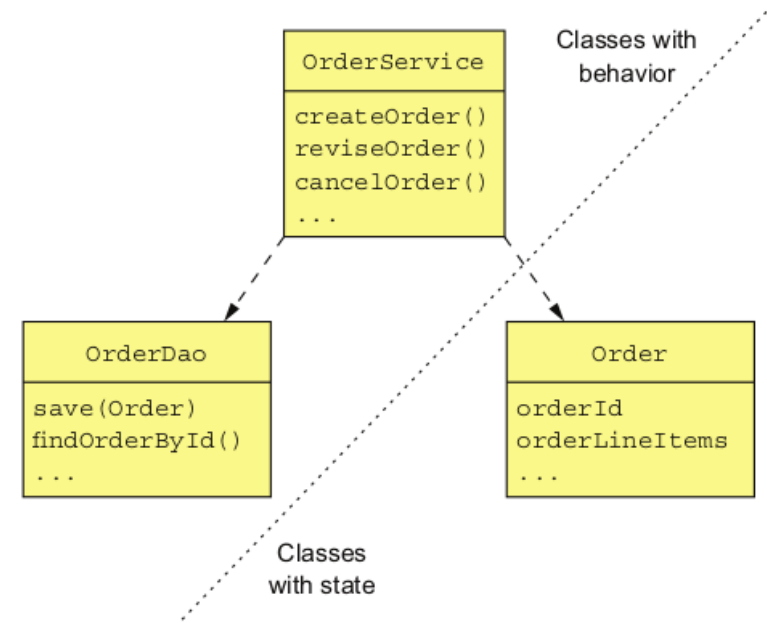
# Transaction Script Pattern

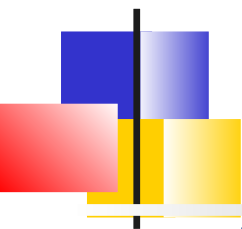
**Transaction Script Pattern**<sup>1</sup> est le plus simple des patterns pour la logique métier.

La logique métier est organisée en procédure. Chaque procédure correspond à une requête possible du système

Les scripts sont typiquement présents dans les classes services

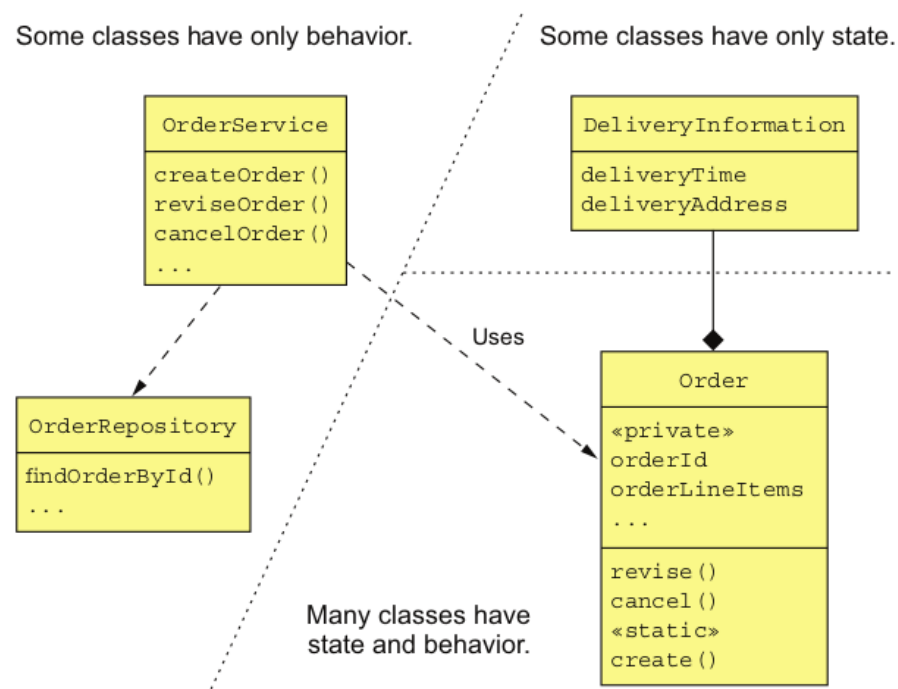
Inconvénient : Si la logique est complexe, on tend vers un code spaghetti





# Domain Model Pattern

**Domain model Pattern**<sup>1</sup> organise la logique métier dans un modèle Objet dont les classes contiennent un état **et** un comportement



1. <https://martinfowler.com/eaCatalog/domainModel.html>



# Avantages du Domain Model pattern

---

Avec ce pattern, les méthodes de la classe service sont généralement simples.

- Elle délègue presque toujours aux objets de domaine qui contiennent l'essentiel de la logique métier.

Le design est facile à comprendre et à maintenir

- Au lieu de se composer d'une grande classe qui fait tout, il se compose d'un certain nombre de petites classes qui ont chacune un petit nombre de responsabilités.

Le design est plus facile à tester

- Chaque classe peut être testée indépendamment

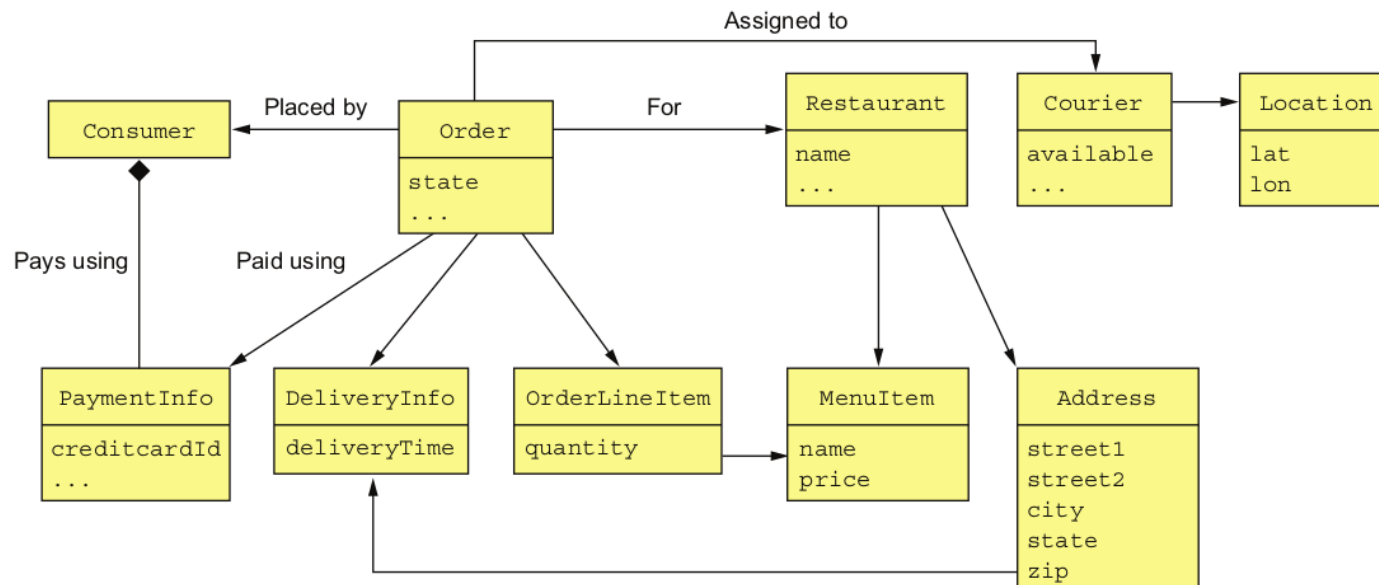
Le design est plus extensible

- On peut appliquer les patterns *Strategy* ou *Template* sur les classes du domaine

# Délimitation du Domain Model

Dans la conception traditionnelle orientée objet, un modèle de domaine est un ensemble de classes et de relations entre les classes. Les classes sont généralement organisées en packages.

Mais quelles classes font partie de l'objet *Order* ?





# Frontières floues

---

L'absence de frontières dans le modèle objet peut créer des problèmes.

Exemple :

- *Order* a un invariant : Montant minimal
- 2 transactions travaillent sur le même *Order*
  - Chacune supprime des items dans l'*Order*
  - Pour chacune des transactions, l'invariant est respecté. La transaction peut se valider
  - Mais au final, on se retrouve avec un *Order* ne respectant plus l'invariant





# Aggregate Pattern

---

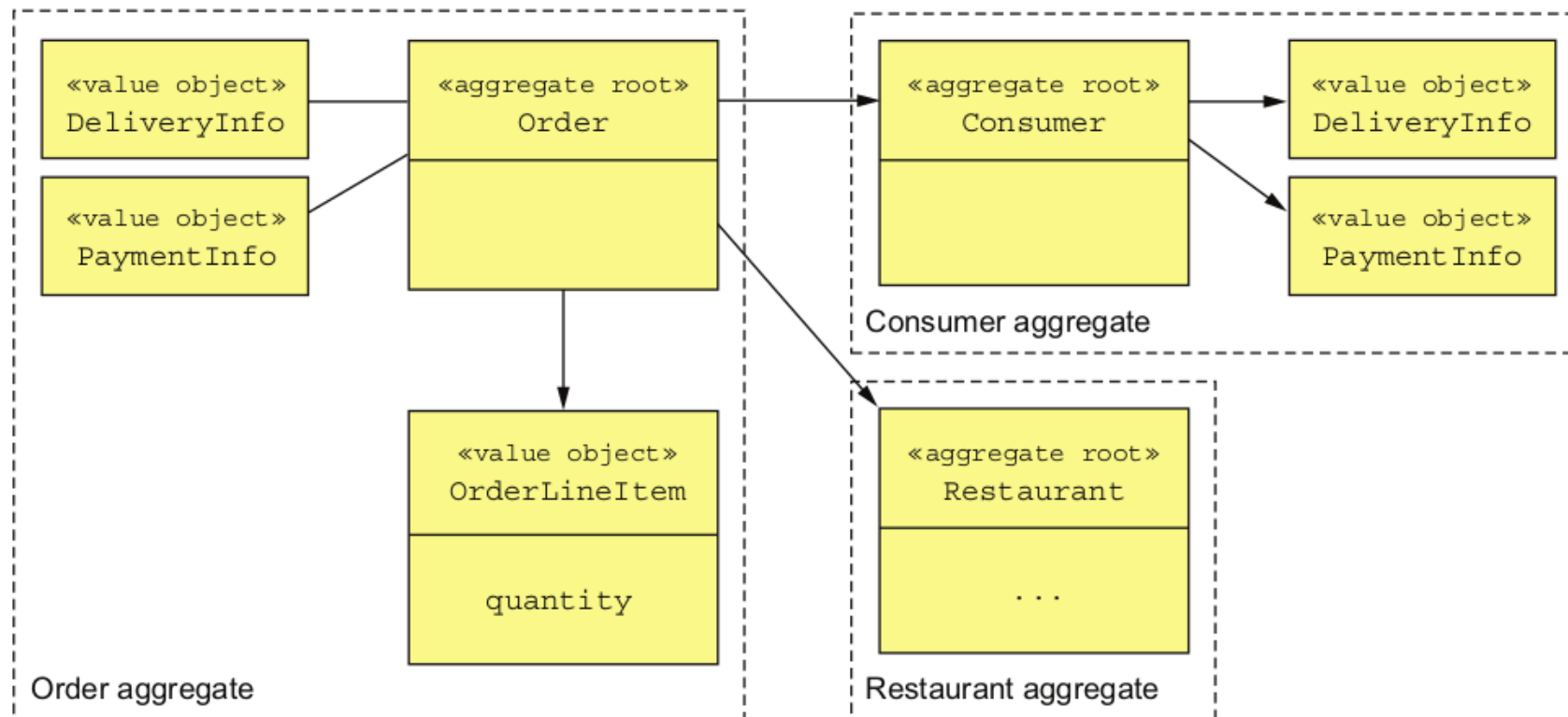
**Aggregate Pattern**<sup>1</sup> : Organise le modèle du domaine comme une collection d'agrégats, i.e. un graphe d'objets traité unitairement

- Les agrégats décomposent un modèle en morceaux, qui sont individuellement plus faciles à comprendre.
- Ils clarifient également la portée des opérations telles que le chargement, la mise à jour et la suppression. Ces opérations agissent sur l'ensemble de l'agrégat.
- La suppression d'un agrégat supprime tous ses objets d'une base de données.

1. [https://martinfowler.com/bliki/DDD\\_Aggregate.html](https://martinfowler.com/bliki/DDD_Aggregate.html)

Voir également <https://blog.engineering.publicissapient.fr/2018/06/25/craft-les-patterns-tactiques-du-ddd/>

# Example





# Avantages

---

La mise à jour d'un agrégat entier plutôt que de ses parties résout les problèmes de cohérence

- Les opérations de mise à jour sont appelées sur l'agrégat racine qui applique les invariants.

La concurrence est gérée en verrouillant l'agrégat racine en utilisant, par exemple, un numéro de version ou un verrou au niveau de la base de données

=> Dans DDD, un élément clé de la conception d'un modèle de domaine consiste à identifier les agrégats, leurs limites et leurs racines.

Leur granularité aura des impacts sur la scalabilité de l'application et également la décomposition en micro-service



# Règles sur les agrégats

---

Les agrégats doivent cependant respecter certaines règles :

1. Référencer seulement l'agrégat racine

Cela garantit que l'agrégat puisse toujours vérifier ses invariants.

2. Les références inter-agrégats utilisent les clés primaires.

L'utilisation d'identité plutôt que de références d'objet signifie que les agrégats sont faiblement couplés

3. UNE transaction crée ou met à jour UN agrégat.

Les opérations devant mettre à jour plusieurs agrégats utilisent SAGA



# Événements métier

---

D'autres parties, (les utilisateurs, les autres services ou les autres composants) sont souvent intéressées par les changements d'états des agrégats.

Par exemple :

- Maintenir la cohérence des données entre les services à l'aide de sagas
- Notifier un service qui gère un réplica que les données source ont changé.
- Notifier une autre application afin de déclencher la prochaine étape d'un processus métier.
- Notifier un composant différent (Par exemple, mettre à jour l'index d'un moteur de recherche)
- Envoi de notifications (messages texte ou e-mails) aux utilisateurs.
- Faire du monitoring ou de l'analyse d'usage



# *Domain Event Pattern*

---

***Domain event Pattern***<sup>1</sup> : Un agrégat publie un événement domaine lorsqu'il est créé ou subit un changement significatif

Un événement domaine est nommé à l'aide d'un verbe au participe passé.

Il a en général

- Des propriétés qui caractérisent l'événement.
- Des méta-données comme un ID et un horodatage
- Quelque fois l'agrégat complet concerné

1. <https://microservices.io/patterns/data/domain-event.html>



# Génération et publication des évènements

---

Les événements doivent être créés puis typiquement publiés vers un message broker.

- La création de l'évènement concerne typiquement l'agrégat (*Domain Model Pattern*)
- Afin que l'agrégat n'ait pas de dépendance sur l'API de messaging, il souhaitable que la classe Service qui peut profiter de l'injection de dépendance publie le message.

Tout cela fait partie d'une transaction locale qui implique la BD et le message Broker  
(*Transactional Outbox Pattern*)



# Exemple

---

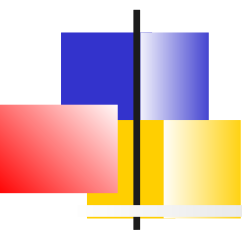
// L'agrégat, instancie l'événement

```
public class Ticket {  
  
    public List<DomainEvent> accept(ZonedDateTime readyBy) {  
        ...  
        this.acceptTime = ZonedDateTime.now();  
        this.readyBy = readyBy;  
        return singletonList(new TicketAcceptedEvent(readyBy));  
    }  
}
```

// Le service connaît l'API de messaging

```
public class KitchenService {  
    @Autowired  
    private TicketRepository ticketRepository;  
    @Autowired  
    private DomainEventPublisher domainEventPublisher;  
    public void accept(long ticketId, ZonedDateTime readyBy) {  
        Ticket ticket = ticketRepository.findById(ticketId).orElseThrow(() ->  
            new NotFoundEx(ticketId));  
        List<DomainEvent> events = ticket.accept(readyBy);  
        domainEventPublisher.publish(Ticket.class, ticketId, events);  
    }  
}
```





# Implémentation avec la POO

---

Architecture logicielle  
Patterns pour la logique métier

## **Couche contrôleur**

Les problématiques de  
sérialisation/désérialisation  
Impacts sur la couche DAO



# Classes contrôleur

---

C'est dans ces classes que les principes RESTFul sont codés :

- Mapping des ressources vers les méthodes
- Récupération des paramètres HTTP
- Positionnement des bonnes entêtes
- Retour du bon status

Le principe de Separation Of Concerns implique que le code des méthodes se concentre sur ces aspects et délègue le reste logique métier, logique de persistance aux couches plus basses



# Recommandations

---

Un contrôleur par ressource REST

Validation des données d'entrées

- Content-type
- Paramètres
- Body

Gestion centralisée des Exceptions, du CORS

Pattern delegate pour isoler le code des annotations OpenAPI



# Spring MVC

---

SpringMVC facilite la mise en place des conventions

- Annotation *@RestController* qui restreint le content-type à JSON
- Annotation *@RequestMapping* au niveau de la classe qui permet de préfixer tous les mappings des méthodes avec le nom de la ressource
- Annotation *@GetMapping*, *@PostMapping*, ... permettant de limiter à une méthode
- Annotation *@Valid* qui permet la validation des données d'entrée via *javax.validation*
- Constructeur de *ResponseEntity* permettant de positionner les codes retours et les entêtes



# Exemple Spring

---

```
@RestController
@RequestMapping("/members")
public class MemberRestController {

    @Autowired private MemberRepository memberRepository;

    @GetMapping("/{id}")
    public Member findOne(@PathVariable Long id) throws MemberNotFoundException {
        return memberRepository.fullLoad(id).orElseThrow(() -> new MemberNotFoundException());
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteOne(Long id) throws MemberNotFoundException {
        Member member = memberRepository.findById(id).orElseThrow(() -> new MemberNotFoundException());
        memberRepository.delete(member);

        return new ResponseEntity<Void>(HttpStatus.NO_CONTENT);
    }

    @PostMapping()
    public ResponseEntity<Member> create(@Valid @RequestBody Member member) {
        member.setRegisteredDate(new Date());
        return ResponseEntity.status(HttpStatus.CREATED)
            .body(memberRepository.save(member));
    }
}
```



# Implémentation avec la POO

---

Architecture logicielle

Patterns pour la logique métier

Couche contrôleur

**Les problématiques de  
sérialisation/désérialisation**

Impacts sur la couche DAO

Exceptions, CORS



# Sérialisation JSON

---

Un des principales problématiques des back-end Spring et la conversion des objets du domaine au format JSON.

Des librairies spécialisés sont utilisées (Jackson, Gson), elles permettent de bénéficier de comportement par défaut

Mais, généralement le développeur doit régler certaines problématiques :

- Boucle infinie pour les relations bidirectionnelles entre classes du modèle
- Adaptation aux besoins de l'interface du front-end
- Optimisation du volume de données échangées
- Format des dates



# Comportement par défaut

---

```
public class Member {  
    private long id;  
    private String nom,prenom;  
    private int age;  
    private Date registeredDate;  
}
```

Devient :

```
{  
    "id": 5,  
    "nom": "Dupont",  
    "prenom": "Gaston",  
    "age": 71,  
    "registeredDate": 1645271583944 // Nombre de ms depuis le 1er Janvier 1970  
}
```





# Concepts Jackson

---

Avec Jackson, les sérialisations/désérialisations sont effectuées généralement par des ***ObjectMapper***

**// Sérialisation**

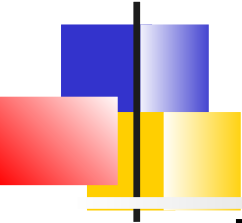
```
Member m = memberRepository.findById(4l) ;  
ObjectMapper objectMapper = new ObjectMapper() ;  
String jsonString = ObjectMapper.writeValueAsString(m) ;
```

...

**// Désérialisation**

```
String jsonString= "{\n\"id\" : 5,\n\" + ... + \"}\" ;  
Member m2 = ObjectMapper.readValue(jsonString) ;
```

Dans un contexte SpringBoot, on utilise rarement l'objet *ObjectMapper* directement ... mais on influence son comportement par des annotations.



# Solutions aux problématiques de sérialisation

---

Pour adapter la sérialisation par défaut de Jackson à ses besoins, 3 alternatives :

- Créer des classes DTO spécifiques.  
La couche *Service* transforme les classes *Entité* provenant de la couche *Repository* en des classes Data Transfer Object encapsulant les données qui sont sérialisées par Jackson
- Utiliser les annotations proposées par Jackson  
Sur les classes DTO ou les classes *Entité*, utiliser les annotations Jackson pour s'adapter au besoin de la sérialisation
- Utiliser l'annotation *@JsonView*  
Le même objet *Entité* ou *Dto* peut alors être sérialisé différemment en fonction des cas d'usage
- Implémenter ses propres Sérialiseur/Désérialiserur.  
Spring propose l'annotation *@JsonComponent*



# Exemple DTO

---

```
@Service
public class UserService {
    @Autowired UserRepository userRepository;
    @Autowired RolesRepository rolesRepository;

    UserDto retrieveUser(String login) {
        User u = userRepository.findByLogin(login);
        List<Role> roles = rolesRepository.findByUser(u);

        return new UserDto(u,roles);
    }
}
```

---

```
public class UserDto {
    private String login, email, nom, prenom;
    List<Role> roles;

    public UserDto(User user, List<Role> roles) {
        login = user.getLogin(); email = user.getEmail();
        nom = user.getNom(); prenom = user.getPrenom();
        this.roles = roles;
    }
}
```



# Format de Dates

---

Pour avoir une représentation String des dates selon les bon vouloir du front-end, la solution est plus souple est d'utiliser @JsonFormat

```
public class Event {  
    public String name;  
    @JsonFormat(shape = JsonFormat.Shape.STRING,  
                pattern = "dd-MM-yyyy hh:mm:ss")  
    public Date eventDate;  
}
```



# Relations bidirectionnelles

## Le problème

---

```
public class User {  
    public int id;  
    public String name;  
    public List<Item> userItems;  
}  
  
public class Item {  
    public int id;  
    public String itemName;  
    public User owner;  
}
```

Lorsque *Jackson* sérialise l'une des 2 classes, il tombe dans une boucle infinie



# Relations bidirectionnelles

## Une solution

---

En annotant les 2 classes avec **@JsonManagedReference** et **@JsonBackReference**

```
public class User {  
    public int id;  
    public String name;  
  
    @JsonManagedReference  
    public List<Item> userItems;  
}
```

```
public class Item {  
    public int id;  
    public String itemName;  
  
    @JsonBackReference  
    public User owner;  
}
```

La propriété *userItems* est sérialisé mais pas *owner*



# Relations bidirectionnelles

## Une autre solution

En annotant les classes avec **@JsonIdentityInfo** qui demande à Jackson de sérialiser une classe juste avec son ID

```
@JsonIdentityInfo(  
    generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")  
public class User {...}
```

```
@JsonIdentityInfo(  
    generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")  
public class Item { ... }
```

Sérialisation d'un Item :

```
{  
  "id":2,  
  "itemName":"book",  
  "owner":  
    {  
      "id":1,  
      "name":"John",  
      "userItems":[2]  
    }  
}
```



# Relations bidirectionnelles

## Une autre solution

---

En annotant les classes avec **@JsonIgnore** on demande à Jackson de ne pas sérialiser une propriété

```
public class User {  
    public int id;  
    public String name;  
  
    public List<Item> userItems;  
}
```

```
public class Item {  
    public int id;  
    public String itemName;  
  
    @JsonIgnore  
    public User owner;  
}
```





# @JsonView

Des relations d'héritages peuvent être définies dans des classes statiques vides

```
public class CompanyViews {  
    public static class Normal{};  
    public static class Manager extends Normal{};  
    public static class HR extends Normal{};  
}
```

Les classes sont ensuite référencées via l'annotation *@JsonView* :

- Sur les classes du modèles :  
Quel attribut est sérialisé lorsque telle vue est activée ?
- Sur les méthodes des contrôleurs :  
Quelle vue doit être utilisée lors de la sérialisation de la valeur de retour de cette méthode ?



# @JsonView

## Annotations sur la classe du modèle

---

```
..  
public class Staff {  
  
    @JsonView(CompanyViews.Normal.class)  
    private String name;  
  
    @JsonView(CompanyViews.Normal.class)  
    private int age;  
  
    // 2 vues  
    @JsonView({CompanyViews.HR.class, CompanyViews.Manager.class})  
    private String[] position;  
  
    @JsonView(CompanyViews.Manager.class)  
    private List<String> skills;  
  
    @JsonView(CompanyViews.HR.class)  
    private Map<String, BigDecimal> salary;  
}
```



# Activation d'une vue

---

```
@RestController
public class StaffController {

    @GetMapping
    @JsonView(CompanyViews.Normal.class)
    public List<Staff> findAll() {
    }

    ...

    ObjectMapper mapper = new ObjectMapper();

    Staff staff = createStaff();

    try {
        String normalView =
            mapper.writerWithView(CompanyViews.Normal.class).writeValueAsString(staff);
```



# Autres annotations Jackson

---

**@JsonProperty, @JsonGetter,  
@JsonSetter, @JsonAnyGetter,  
@JsonAnySetter, @JsonIgnore,  
@JsonIgnoreProperty, @JsonIgnoreType :**  
Permettant de définir les propriétés JSON

**@JsonRootName :** Arbre JSON

**@JsonSerialize, @JsonDeserialize :** Indique  
des dé/sérialiseurs spécialisés

....



# Sérialiseur spécifique

---

L'annotation *Spring* **@JsonComponent** facilite  
l'enregistrement de sérialiseurs/désérialiseurs Jackson

Elle doit être placée sur des implémentations de *JsonSerializer*  
et *JsonDeserializer* ou sur des classes contenant des inner-  
class de ce type

## **@JsonComponent**

```
public class Example {  
    public static class Serializer extends JsonSerializer<SomeObject> {  
        // ...  
    }  
    public static class Deserializer extends  
JsonDeserializer<SomeObject> {  
        // ...  
    }  
}
```



# Implémentation avec la POO

---

Architecture logicielle

Patterns pour la logique métier

Couche Controller

Les problématiques de  
sérialisation/désérialisation

**Impacts sur la couche JPA**

Exceptions, CORS



# Introduction

---

Au moment de la sérialisation, la librairie de sérialisation appelle les différents getter sur la classe du modèle.

La couche JPA introduit généralement des mécanismes de chargements lazy (principalement pour les associations *@OneToMany*)

Le développeur peut alors être confronté à des *LazyInitialisationException*, i.e. Appel d'un getter sur une association qui n'a pas été chargée après la fermeture de l'entity Manager

2 solutions classiques :

- Le pattern Open Entity Manager in View
- S'assurer de précharger les associations avant la sérialisation



# OpenInView

Le pattern “**Open EntityManager in View**” synchronise le cycle de vie de l’entity Manager avec le traitement de la requête

- Spring Boot l’implémente via l’intercepteur *OpenEntityManagerInViewInterceptor*  
C’est le comportement par défaut, pour le désactiver :  
`spring.jpa.open-in-view = false`
- Lors de la sérialisation JSON, il est alors possible de faire des requêtes SQL

Ce pattern est quelquefois considéré comme un anti-pattern :

- Les requêtes de chargement ne sont pas optimisées
- Les connexions BD sont maintenues trop longtemps





# Pré-chargement des associations

---

L'autre solution pour éviter les LazyException est de s'assurer du chargement des associations voulues par le cas d'usage.

Différentes techniques sont possible avec JPA :

- Utilisation d'une query JPQL avec jointure
- Méthode statique Hibernate.initialize
- Utilisation des EntityGraph



# Implémentation avec la POO

---

Architecture logicielle  
Patterns pour la logique métier  
Couche Controller  
Les problématiques de  
sérialisation/désérialisation  
Impacts sur la couche JPA  
**Exceptions, CORS**



# Personnalisation de la configuration Spring MVC

---

- Le personnalisation de la configuration par défaut de *SpringBoot* peut être effectuée en définissant un bean de type ***WebMvcConfigurer*** et en surchargeant les méthodes proposée.
- Dans le cadre d'une API Rest, une méthode permet de configurer le CORS<sup>1</sup>

1. CORS : *Cross-origin resource sharing*, une page web ne peut pas faire de requêtes vers d'autre serveurs que son serveur d'origine.



# Cross-origin

---

Le CORS peut se configurer globalement en surchargeant la méthode *addCorsMapping* de *WebMvcConfigurer* :

```
@Configuration
public class MyConfiguration implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**").allowedOrigins("*");
    }
}
```

A noter qu'il est également possible de configurer le cors individuellement sur les contrôleurs via l'annotation **@CrossOrigin**



# Gestion des erreurs

---

*Spring Boot* associe ***/error*** à la page d'erreur globale de l'application

- Un comportement par défaut en REST ou en Web permet de visualiser la cause de l'erreur dans une page HTML

Pour remplacer le comportement par défaut (Par exemple générer un JSON d'erreur) :

- L'annotation ***ResponseStatus*** sur une exception métier susceptible d'être lancée par un contrôleur
- Lancer une ***ResponseStatusException*** permettant de positionner le code retour
- Ajouter un bean de type ***@ControllerAdvice*** qui hérite de *ResponseEntityExceptionHandler* pour centraliser la génération de réponse lors d'exception



# Exemple

---

**@ResponseStatus(value = HttpStatus.NOT\_FOUND)**

```
public class MyResourceNotFoundException extends RuntimeException {  
    public MyResourceNotFoundException() {  
        super();  
    }  
    public MyResourceNotFoundException(String message, Throwable cause) {  
        super(message, cause);  
    }  
    public MyResourceNotFoundException(String message) {  
        super(message);  
    }  
    public MyResourceNotFoundException(Throwable cause) {  
        super(cause);  
    }  
}
```



# *ResponseStatusException*

---

```
@GetMapping(value =("/{id}")
public Foo findById(@PathVariable("id") Long id, HttpServletResponse response)
{
    try {
        Foo resourceById = RestPreconditions.checkFound(service.findOne(id));

        eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this,
response));
        return resourceById;
    }
    catch (MyResourceNotFoundException exc) {
        throw new ResponseStatusException(
            HttpStatus.NOT_FOUND, "Foo Not Found", exc);
    }
}
```



# Exemple *@ControllerAdvice*

## **@ControllerAdvice**

```
public class NotFoundAdvice extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = {MemberNotFound.class, DocumentNotFound.class})
    ResponseEntity<Object> handleNotFoundException(HttpServletRequest request,
        Throwable ex) {
        return new ResponseEntity<Object>(
            "Entity was not found", new HttpHeaders(), HttpStatus.NOT_FOUND);
    }

    @Override
    protected ResponseEntity<Object>
        handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<Object>(
            ex.getMessage(), new HttpHeaders(), HttpStatus.BAD_REQUEST);
    }
}
```





# Sécurité

---

## **OWASP et vulnérabilités**

Authentication et autorisations  
oAuth2/Open ID et JWT



# Introduction

---

**OWASP : Open Web Application Security Project** est une organisation à but non lucratif fondée en 2001

Référence incontestée dans le monde de la sécurité des systèmes d'information

Publie des recommandations, méthodes et outils pour sécuriser les applications web et APIs

Les projets les plus connus :

- *Top Ten OWASP* : liste des dix risques de sécurité applicatifs Web les plus critiques
- *WebGoat* : plateforme de formation
- *WebScarab* : proxy d'audits de sécurité.

**API Security Top 10** : Top 10 des vulnérabilités dans les API



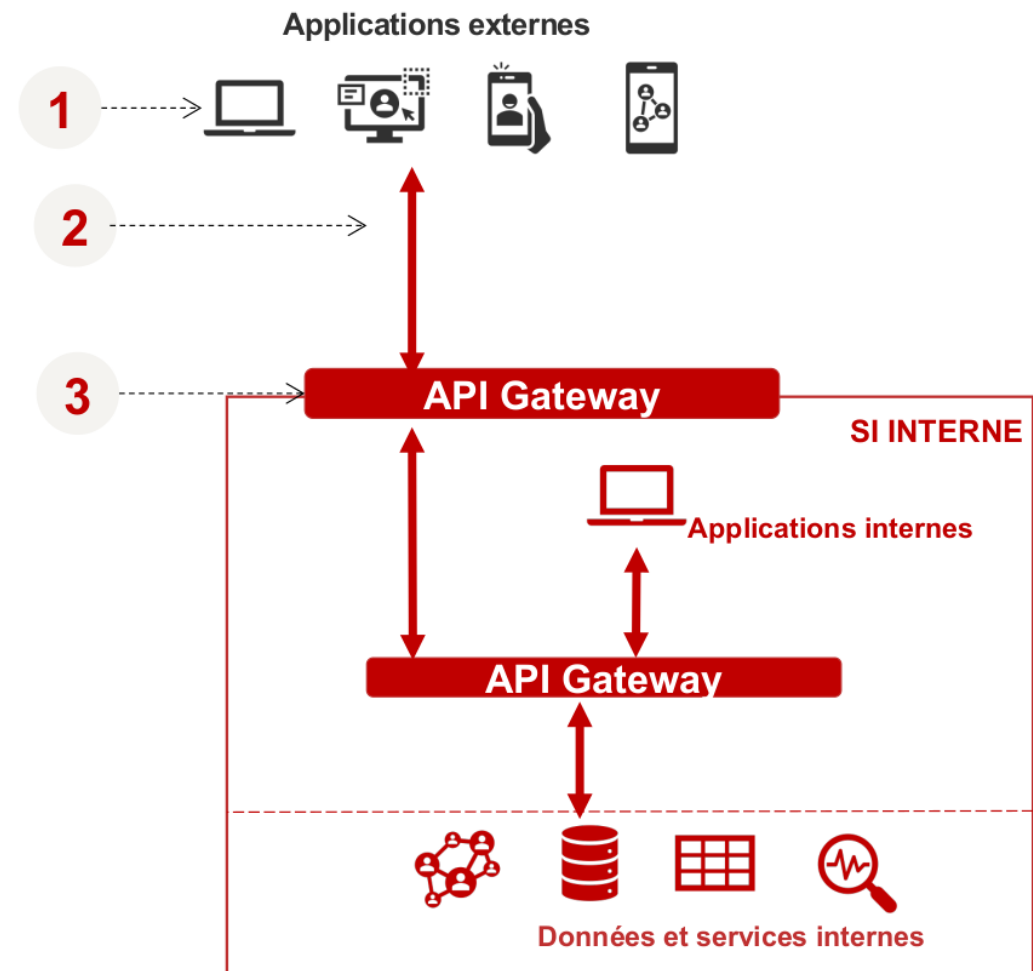
# API Security Top 10

1. **Broken Object Level Authorization** : Contournement des mécanismes d'autorisation pour accéder à un objet non autorisé
2. **Broken Authentication** : Authentification faible permettant à un utilisateur de se faire passer pour quelqu'un d'autre
3. **Excessive data exposure** : Exposer beaucoup plus de données que ce dont le client a besoin
4. **Lack of resources and rate limiting** : une API n'est pas protégée contre un nombre d'appel excessif
5. **Broken function level authorization** : Contrôles d'autorisation insuffisants (exemple: contrôles faits uniquement côté client)
6. **Mass assignment** : Manque de contrôle sur la modification des attributs d'un objet (exemple : rôles, mot de passe)
7. **Security misconfiguration** : Une mauvaise configuration des serveurs, des autorisations ou de l'API permet aux attaquants de les exploiter
8. **Injection** : Des requête qui incluent des commandes (SQL, NoSQL, LDAP, JSON ou autres) malveillantes
9. **Improper assets management** : Exploitations des endpoints obsolètes ou peu sécurisés
10. **Insufficient logging and monitoring** : Le manque de monitoring, de surveillance et d'alertes appropriées permet aux attaquants de passer inaperçus.

# Provenances des attaques

Les attaques peuvent se produire à 3 niveaux

1. Au niveau des applications consommatrices des APIs
  - Vol des credentials, ...
2. Au niveau d'une API
  - Données sensibles, injection de code, ...
3. Au niveau Infra/plateforme API Gateway
  - Accès aux fonctionnalités d'administration, ...





# Comment y remédier ?

---

## 1 . Un socle technique solide

- Sécuriser l'infra/plateforme API Gateway
- Apporter un socle technique commun avec des solutions d'authentification éprouvées

## 2 . De la sécurité by Design

- Classifier les API selon leurs usages (APIs internes, partenaires, open)
- Se poser la question de la sensibilité des données lors de la conception d'une API

## 3 . Des équipes sensibilisés

- Sensibiliser les équipes aux enjeux sécuritaires
- Les former aux socles techniques/outils



# Socle Technique

## Portail d'entrée

---

Le WAF (Web Application Firewall) protège le SI des attaques par

- Dénis de services
- Injection de code (XML, Json, SQL,...)
- Cross-Site Scripting (XSS)
- XML External Entities (XXE)
- Attaque par force brute

Il permet également le filtrage IP et fait office de terminaison SSL/TLS entre l'extérieur et l'API Gateway

**HTTPS pour les échanges avec l'extérieur**



# Socle Technique

## Une Gateway sécurisée

---

Une API Gateway configurée pour un maximum de sécurité

- Rate Limiting
- Gestion des quotas
- Validation des entrées/sorties

Des solutions communes éprouvées d'identification/authentification

- Contrôle d'accès des applications
- Authentification des utilisateurs
- Habilitations fonctionnelles des utilisateurs



# La sécurité by Design

## Gestion des données sensibles

---

1. Séparation des environnements  
Séparation stricte des environnements de test et de production
2. Stockage et gestion des secrets  
Les secrets doivent être stockés et communiqués d'une façon sécurisée (Chiffrement des données).
3. Sécurité by Design
  - Seules les données nécessaires doivent être communiquées dans les réponses
  - Adapter le niveau de sensibilité des données au public cible





# Exemples de bonnes pratiques

---

Identifiant non devinables : Il est recommandé que les Ids des ressources soient non devinables

- éviter les incrémentations par défaut.

Gérer les exceptions d'une façon à masquer les environnements backend et les technologies utilisées

S'interdire l'utilisation de cookies

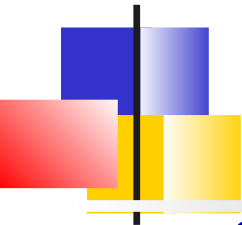


# Surveillance

---

Installez un système d'alarmes :  
composant de surveillance basé sur de  
l'intelligence artificielle pour

- connaître les comportements du trafic
- Détecter les anomalies
- Lever les alerte
- Ou bloquer automatiquement les menaces



**CORS** : Le « Cross-origin resource sharing » (CORS) permet d'ouvrir son API.

=> Gérer les hôtes ou domaines autorisés

**CSRF (Cross-Site Request Forgery)** : Concerne plus les applications Web. Les jetons des APIs Rest protège des CSRF

**Anti-farming** : Protection des données contre le scarpping en détectant les patterns d'une ou plusieurs adresses IP

**Rate-limiting** : Protection l'utilisation d'une API en limitant le nombre de requêtes simultanées (secondes/minutes)

**Throttling** : Limiter le nombre de requêtes qu'une application peut faire dans une période donnée (heures/jours/Mois/Années), i.e. Quota



# Outils de détection

---

## Commercial

- Appspider
- Netsparker
- Acunetix
- Burpsuite
- WebInspect
- WebCruiser

## OpenSource

- Zed Attack Proxy
- Arachni
- IronWASP
- WATOBO



# Sécurité

---

OWASP et vulnérabilités  
**Authentication et autorisations**  
oAuth2/Open ID et JWT



# Authentification et permissions

---

## 1 . Identité des applications

- Une API peut être consommée par plusieurs applications, Il faut identifier chaque application
- Chaque application a son identifiant qui peut être révoqué à tout moment
- Le contrôle est fait par la Gateway

## 2 . Authentification des utilisateurs

- L'accès aux ressources est conditionné par la présentation d'une preuve d'authentification valide
- Le contrôle est fait par l'API Gateway

## 3 . Autorisation

- Une fois authentifié, vérifier que l'application ou l'utilisateur est habilité à accéder à la ressource.
- Le contrôle est fait par l'API Gateway ET le fournisseur backend



# Technologies

---

## Identification

- API Key

## Authentication et autorisation :

- SAML (Security assertion markup language)
- Certificat
- JSON Web Token
- **OAuth 2.0 & OpenID Connect**



# Granularité des permissions

---

## 3 niveaux d'autorisation

- Niveau ressource : autorisation d'accès à la ressource.
- Niveau verbe : méthodes autorisées sur la ressource (create / read / update / delete).
- Niveau propriété : gestion de l'autorisation par propriété (read / write / mask / restricted values per role).





# Sécurité

---

OWASP et vulnérabilités  
Authentification et autorisations  
**oAuth2/Open ID et JWT**



# Rôles du protocole

---

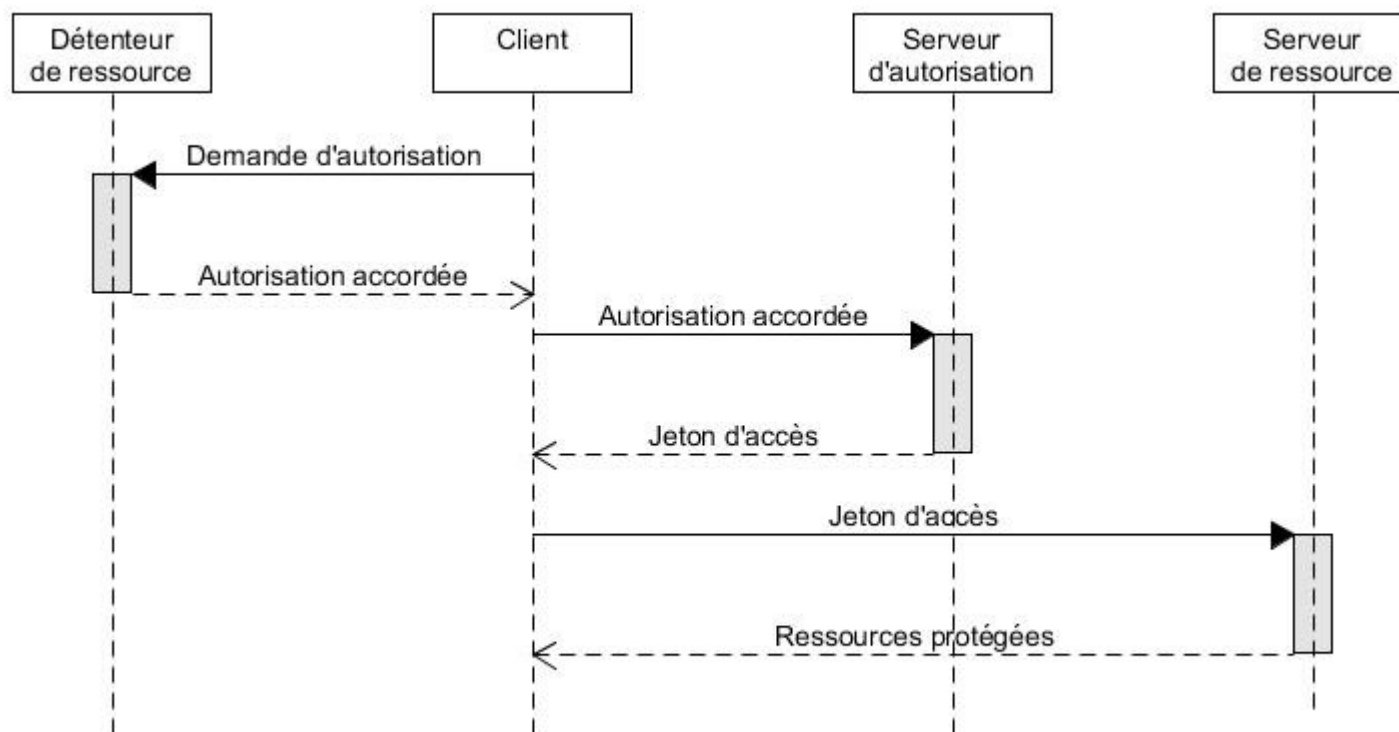
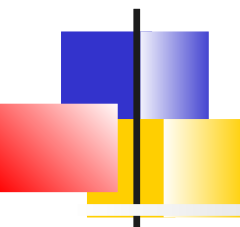
Le **Client** est l'application qui essaie d'accéder au compte utilisateur. Elle a besoin d'obtenir une permission de l'utilisateur pour le faire.

Le **serveur de ressources** est l'API utilisée pour accéder aux ressources protégées

Le **serveur d'autorisation** est le serveur qui autorise un client à accéder aux ressources en lui fournissant un jeton. Il peut demander l'approbation de l'utilisateur

**L'utilisateur** est la personne qui donne accès à certaines parties de son compte

Rq: Un participant du protocole peut jouer plusieurs rôles





# Scénario

---

1. Pré-enregistrer le client auprès du service d'autorisation (=> client ID et un secret)
2. Obtenir l'autorisation de l'utilisateur.  
(4 types de grant)
3. Obtention du token (date d'expiration)
4. Appel de l'API pour obtenir les informations voulues en utilisant le token
5. Validation du token par le serveur de ressource



# Enregistrement du client

---

Le protocole ne définit pas comment l'enregistrement du client doit se faire mais définit les paramètres d'échange.

Le client doit fournir :

- **Application Name**: Le nom de l'application
- **Redirect URLs**: Les URLs du client pour recevoir le code d'autorisation et le jeton d'accès
- **Grant Types** : Les types d'autorisations utilisables par le client
- **Scopes** : paramètre utilisé pour limiter les droits d'accès d'un client
- **Javascript Origin** (optionnel): Le host autorisé à accéder aux ressources via *XMLHttpRequest*

Le serveur répond avec :

- **Client Id**:
- **Client Secret**: Clé devant rester confidentielle



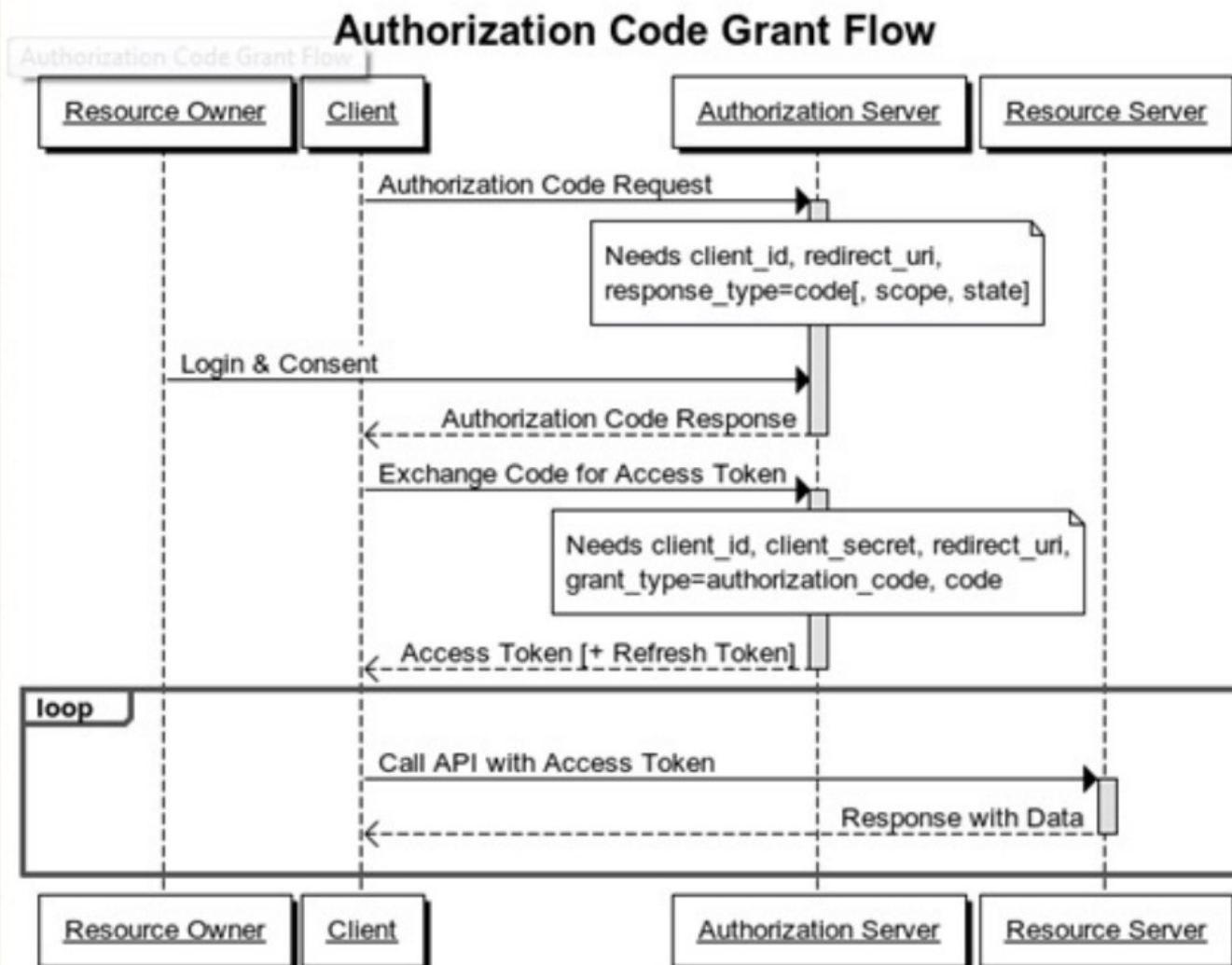
# *OAuth2 Grant Type*

---

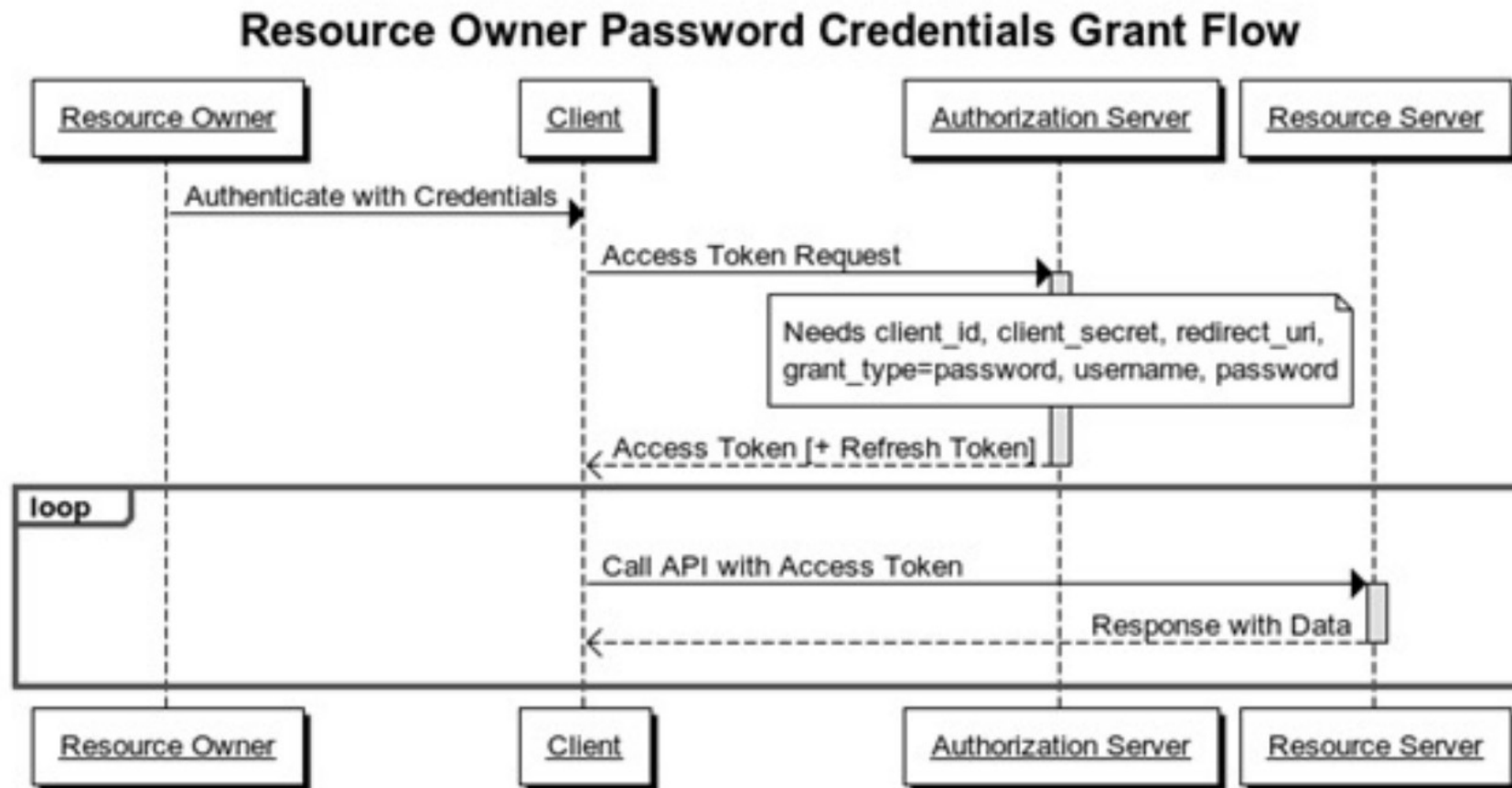
Différents moyens afin que l'utilisateur donne son accord : les **grant types**

- ***authorization code*** : Approbation de l'utilisateur sur le serveur d'autorisation et échange d'un code d'autorisation avec le client
- ***implicit*** : Jeton fourni directement. Certains serveurs interdisent de mode
- ***password*** : Le client fournit les créidentiels de l'utilisateur
- ***client credentials*** : Les créidentiels client suffise

# Authorization Code

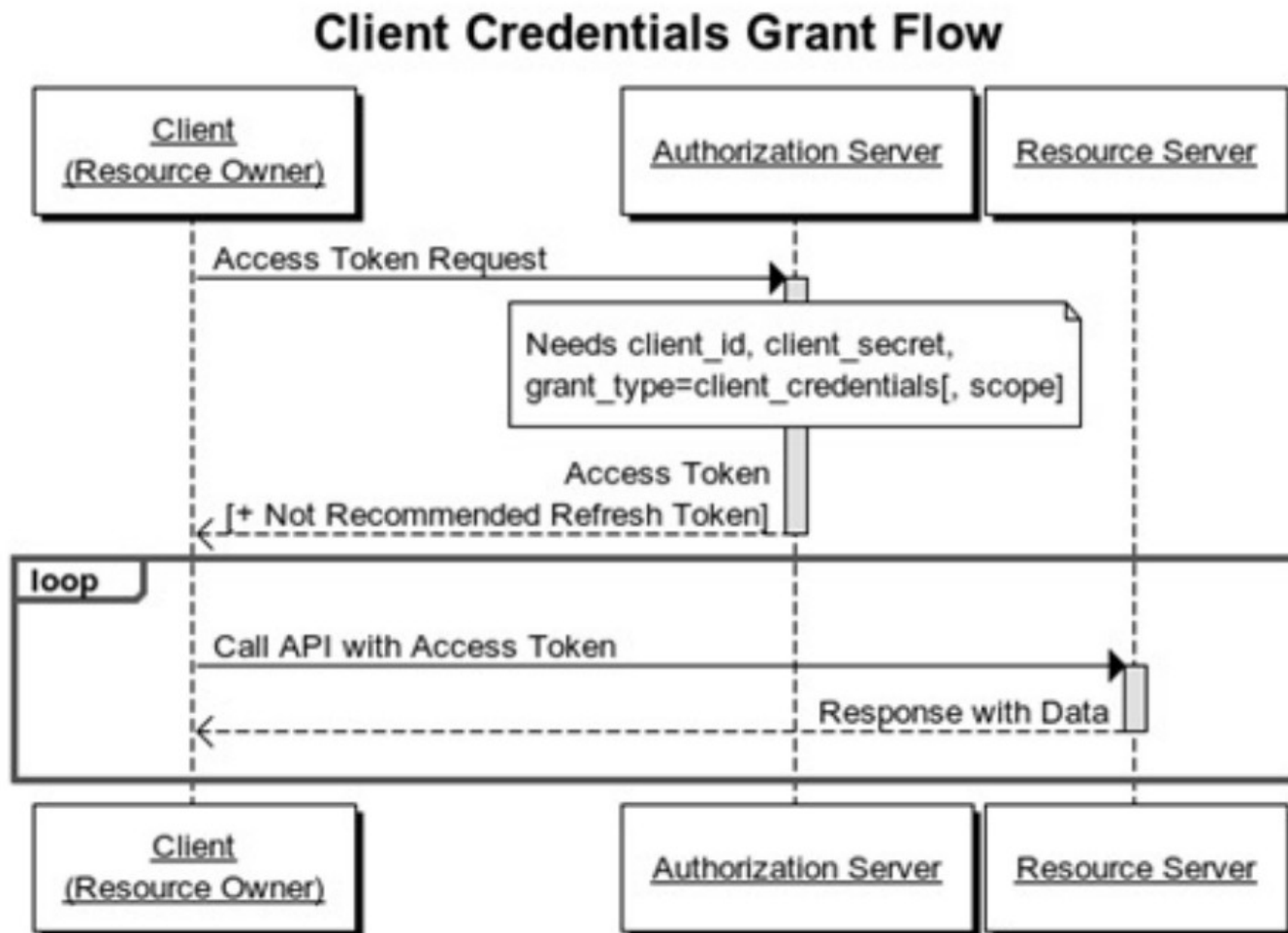


# Password Grant





# Client Credentials





# Tokens

---

Les Tokens sont des chaînes de caractères aléatoire générées par le serveur d'autorisation

Les jetons sont ensuite présents dans les requêtes HTTP et contiennent des informations sensibles => HTTPS

Il y a 2 types de token

- Le **jeton d'accès**: Il a une durée de vie limité.
- Le **Refresh Token**: Délivré avec le jeton d'accès. Il est renvoyer au serveur d'autorisation pour renouveler le jeton d'accès lorsqu'il a expiré



# Usage du jeton

---

Le jeton est passé à travers 2 moyens :

- Les paramètres HTTP. (Les jetons apparaissent dans les traces du serveur)
- ***L'entête d'Authorization***

```
GET /profile HTTP/1.1
```

```
Host: api.example.com
```

```
Authorization: Bearer MzJmNDc3M2VjMmQzN
```

<http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/>



# Validation du jeton

---

Lors de la réception du jeton, le serveur de ressource doit valider l'authenticité du jeton et extraire ses informations différentes techniques sont possibles

- Appel REST vers le serveur d'autorisation
- Utilisation d'un support persistant partagé (ex. JdbcStore)
- Utilisation de JWT et validation via clé privé ou clé publique



# JWT

**JSON Web Token (JWT)** est un standard ouvert défini dans la RFC 75191.

Il permet l'échange sécurisé de jetons (tokens) entre plusieurs parties.

La sécurité consiste en la vérification de l'intégrité des données à l'aide d'une signature numérique. (HMAC ou RSA).

Dans le cadre d'une application REST SpringBoot, le jeton contient les informations d'authentification d'un user :  
Subject + Rôles

Différentes implémentations existent en Java (*io.jsonwebtoken*, ...) ou le starter ***spring-security-oauth2-jose***



# Configuration typique *SpringBoot*

---

## @Configuration

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors()
            .and()
            .authorizeRequests()
                .antMatchers(HttpMethod.GET, "/user/info", "/api/foos/**")
                    .hasAuthority("SCOPE_read")
                .antMatchers(HttpMethod.POST, "/api/foos")
                    .hasAuthority("SCOPE_write")
                .anyRequest()
                    .authenticated()
            .and()
                .oauth2ResourceServer()
                    .jwt();
    }
}
```