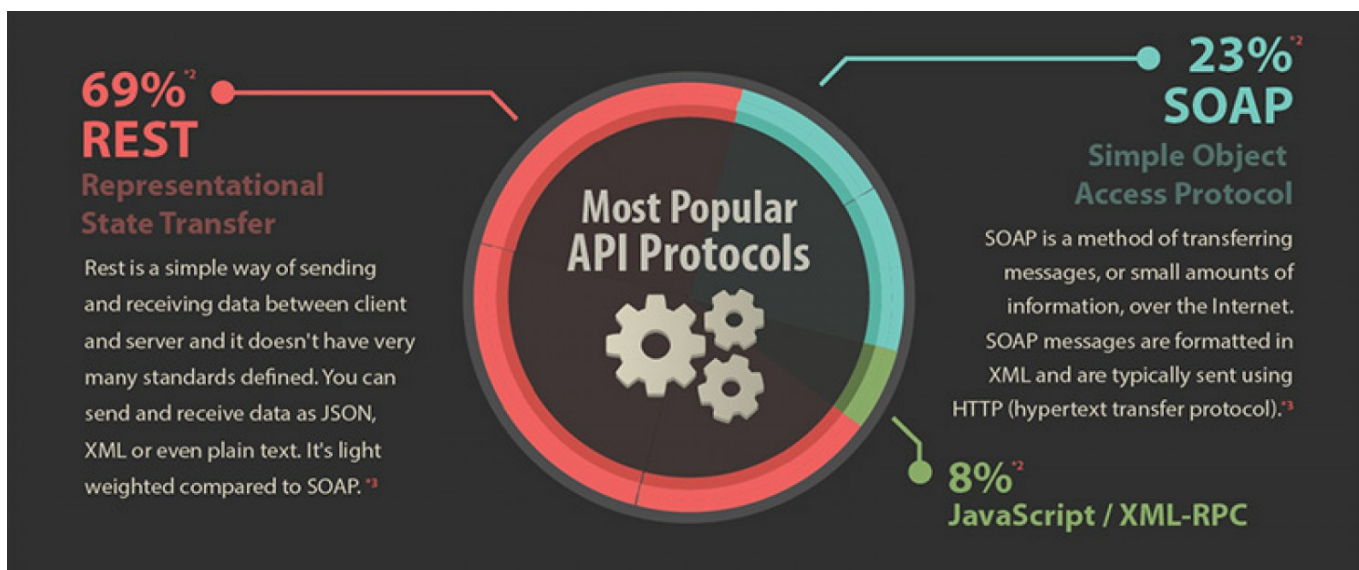


# Fondamentaux REST

Les API REST - Representational State Transfert - sont de plus en plus populaires :



REST est exposé par Roy Thomas Fielding dans sa thèse de Doctorat de philosophie en information et informatique intitulé « [Architectural styles and the design of network-based software architectures](#) » publiée en 2000.

Il a participé dès 1994 à la spécification du World Wide Web par sa participation aux groupes de travail sur URI, HTML et HTTP au sein de l'IETF (Internet Engineering Task Force). Il est également un des co-fondateurs du projet de serveur HTTP Apache et est membre de l'Apache Software Foundation.

REST est une abstraction des éléments architecturaux d'un système réparti d'hypermédias. Ce modèle d'architecture s'appuie sur la description première de l'architecture du World Wide Web :

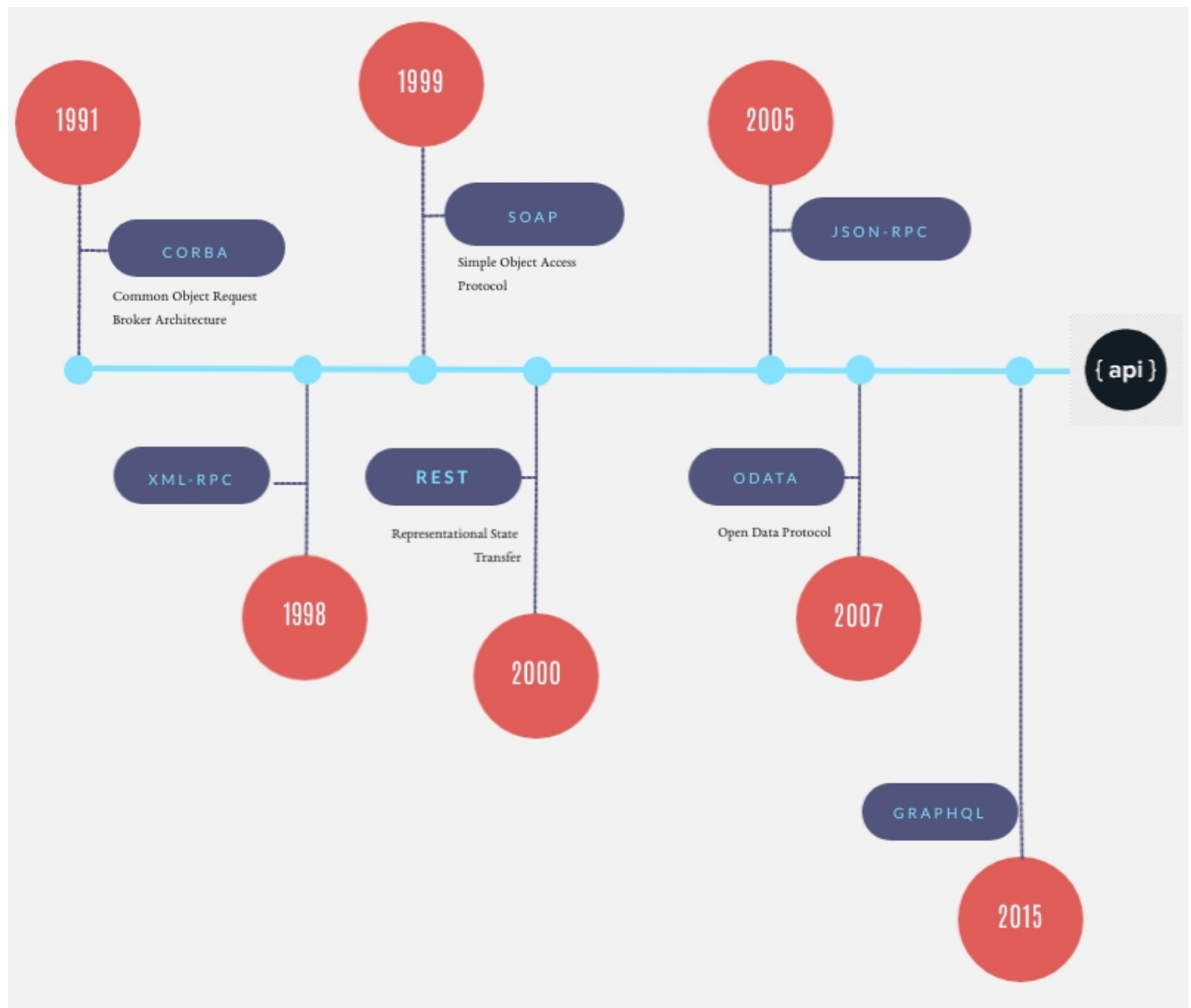
- modèle architectural client / serveur ;
- communication sans état ;
- possibilité de mettre en cache les réponses à des requêtes ;

Et y ajoute de nouvelles contraintes :

- une **interface uniforme** entre les différents composants de l'architecture caractérisée par :
  - l'**identification des ressources** ;
  - la **manipulation des ressources par des représentations** ;
  - des **messages auto descriptifs** ;
  - l'**hypermédia comme "moteur"** de l'application.
- un système en couches : **chaque composant ne peut « voir » au-delà de la couche immédiate avec laquelle il interagit** ;
- un modèle de code à la demande (facultatif) permettant l'extension des fonctionnalités d'un client par le biais de téléchargement et d'exécution de code sous forme d'applet ou de scripts.

**REST n'est pas un standard, il n'existe donc pas de spécification** (cf. [REST - Semantic Web Standards a discussion draft, with no formal status at W3C](#)). Il faut comprendre ce style d'architecture, puis concevoir des

services web (ou des applications web) selon ce style.



Parmi les géants du Web, ce sont eBay, suivi par Amazon qui ont publié les premiers des APIs selon ce modèle (REST), puis Flickr...

## 1. Les principes généraux

Bien que REST ne soit pas un standard, il utilise des standards, en particulier :

- [URI](#) comme syntaxe universelle pour adresser les **ressources** ;
- HTTP ([RFC7230](#), [RFC7231](#), [RFC7232](#), [RFC7233](#), [RFC7234](#), [RFC7235](#), [RFC7236](#), [RFC7237](#)) ;
- Les liens hypermédia ;
- les [types MIMES](#) ;
- ...

### 1.1 L'identification des ressources

L'identification des ressources est une question centrale dans la conception des API REST. Elle doit donc être clairement définie et partagée par tous. Ce sont ses règles d'identification qui vont garantir notamment la cohérence et l'homogénéité des APIs. Cet aspect est d'autant plus important lorsque lesdites APIs deviennent publiques.

### 1.1.1 Noms de domaines des API

La publication de sous-domaines pour les APIs est surtout à prendre en compte lorsque celles-ci sont **publiques**. Par anticipation, il est possible de mettre en place les domaines suivants :

- **production** : `https://api.example.com`
- **tests** : `https://api.sandbox.example.com`

Pour ces deux premiers sous-domaines, il est possible, lorsque les domaines métiers sont multiples et avec des APIs nombreuses, de les distinguer par domaines métiers, mais jamais par domaines techniques !

- **portail développeurs** : `https://developers.example.com`

### 1.1.2 Versionning

La définition de l'architecture REST ne comporte aucun élément quant au versioning des APIs. Le versioning reste cependant un thème à part entière, notamment pour gérer l'évolutivité des services et donc des APIs proposées.

Il existe plusieurs solutions, communément admises :

- **Positionner le numéro de version de l'API, sur un digit (version majeure), au plus haut niveau du chemin de l'URI ;**
- Utiliser les en-têtes HTTP :
  - *Accept* : en utilisant la paramètre *level* Exemple :

```
Accept: application/json;level=1
```

- Utiliser un en-tête personnalisé, *Accept-version*. Exemple :

```
Accept-version: v1
```

Pour sa simplicité de mise en oeuvre, c'est la **première solution** qui est très majoritairement utilisée. La numérotation retient **uniquement le numéro de version majeure**. Il faut donc définir les opérations qui sont à l'origine de ce changement de version, le plus souvent, il s'agit de **breaking changes**, notamment tout changement qui impacte directement les clients existants.

### 1.1.3 Casse

Il existe 3 types principaux de style de casse :

- **CamelCase**, décliné en **lowerCamelCase** et **UpperCamelCase**,
- **snake\_case** (utilisation du **underscore** pour séparer les termes), et
- **spinal-case** (utilisation du **tiret - hyphen** pour séparer les termes).

### 1.1.3.1 URI

La [RFC 3986](#) définit **les URIs sensibles à la casse**, sauf pour le protocole (*scheme*) et l'hôte dans la partie domaine (*authority*).

**Il faut donc préférer l'écriture des URIs en minuscules.**

Pour séparer les termes d'une expression, il reste possible d'utiliser soit l'underscore (*snake\_case*), soit le tiret (*spinal-case*).

Pour une simple question de facilité de lecture et de visibilité des caractères, **il est préférable d'utiliser** le tiret, donc **la notation spinal-case**.

### 1.1.3.2 Corps des requêtes et des réponses

Sur la base des règles, imposées ou de fait, d'un grand nombre de langage de programmation, dont Java, JavaScript, etc., il est recommandé d'utiliser la notation lowerCamelCase.

### 1.1.4 Noms vs. Verbes

Il s'agit ici d'identifier des ressources et non des actions sur lesdites ressources par opposition aux systèmes RPC, voire SOAP... Il convient donc d'utiliser des noms et non des verbes. Il faut utiliser des noms significatifs et éviter les acronymes. Les actions effectuées sur ces mêmes ressources seront portées par la méthode HTTP utilisée par la requête, en particulier pour les opérations CRUD :

Operation	méthode HTTP	Commentaire(s)
Création	POST	Pour créer une instance dans une collection si l'identifiant de l'instance est inconnu. L'identification (l'URI) de la ressource nouvellement créée figure dans l'attribut <i>Location</i> : de la réponse HTTP
Création	PUT	Uniquement lorsque l'identifiant de la nouvelle ressource est fourni par le client dans la requête HTTP
Recherche	GET	Pour la recherche d'une collection ou d'une instance dans une collection
Mise à jour	PUT	Pour une mise à jour complète
Mise à jour	PATCH	pour une mise à jour partielle
Suppression	DELETE	Pour une suppression

Il est malgré tout possible qu'il reste dans les APIs des logiques d'opérations. Dans ce dernier cas, il convient d'utiliser une requête *POST* et de considérer une URI qui se terminera par le verbe identifiant l'opération à effectuer.

### 1.1.5 Singulier vs. Pluriel

Quelle que soit la règle mise en place, elle doit être appliquée systématiquement pour faciliter :

- l'*explorabilité* de l'API, et

- sa logique naturelle de lecture

Il est donc recommandé d'utiliser les noms au pluriel.

Exemple : `/countries/{id}` renvoie un pays (instance) choisi dans la liste (collection) des pays

### 1.1.6 Structure hiérarchique

La profondeur de la structure hiérarchique ne devrait pas dépasser deux niveaux, aussi bien au niveau des URLs d'identification des ressources, qu'au niveau des corps de requêtes et de réponses.

Exemple : URI :

```
/countries/DEU/regions  
/countries/DEU/regions/Saarland
```

Corps (Json)

```
{  
  "shortName": "Germany",  
  "fullName": "Germany",  
  "alpha2": "DE",  
  "alpha3": "DEU",  
  "numeric": "276",  
  "regions": [  
    {  
      "name": "Saarland",  
      (...)  
    },  
    (...)  
  ]  
}
```

---

## En résumé

Ces recommandations permettent d'identifier et de manipuler les ressources dans une grande majorité de cas.

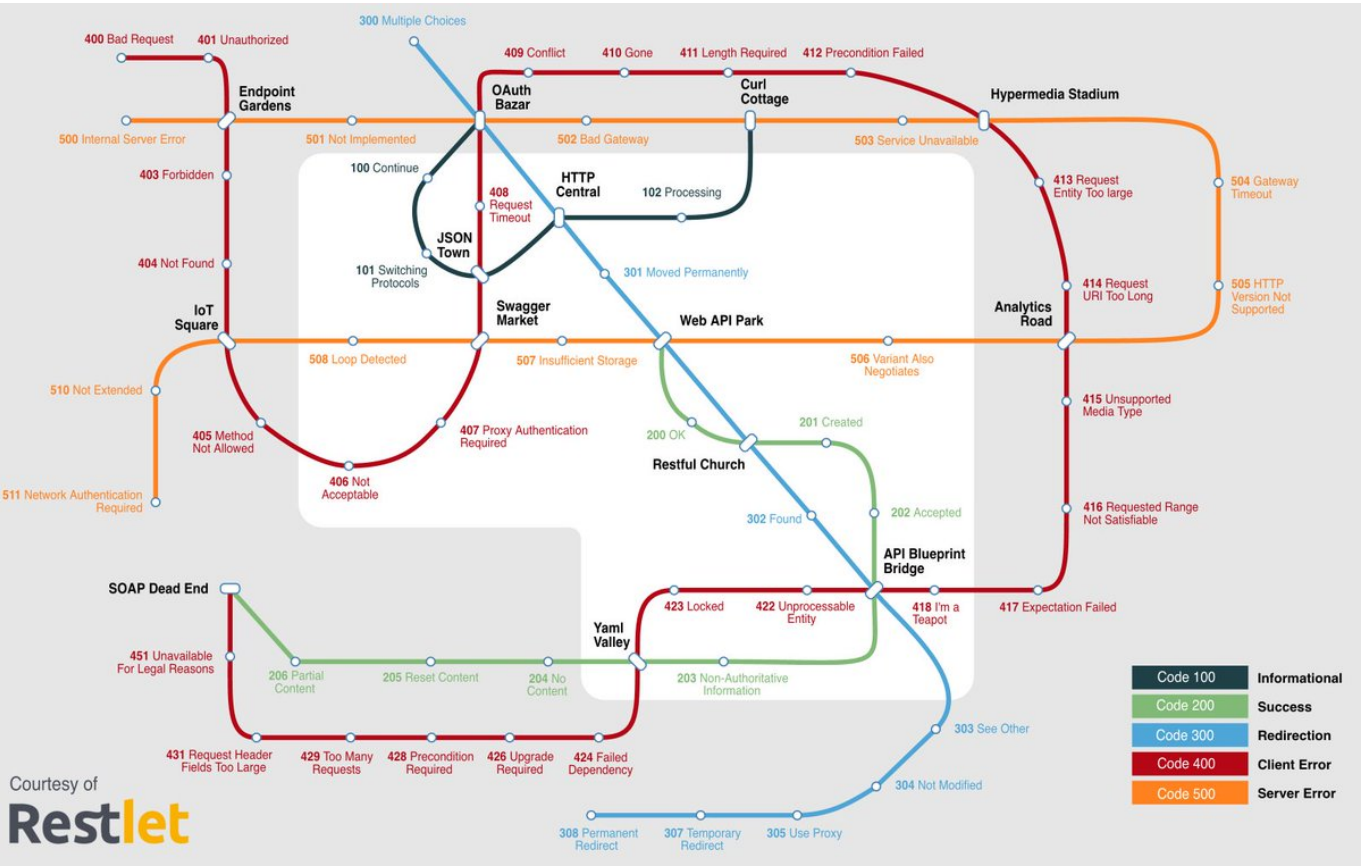
Exemple d'URI selon les principes énoncés :

```
https://api.example.com/v1/countries/DEU/regions  
https://api.example.com/geolocation/v1/countries/DEU/regions  
https://api.geolocation.example.com/v1/countries/DEU/regions
```

Quelle que soient les règles, il faut les appliquer systématiquement pour assurer la cohérence, l'homogénéité, l'"explorabilité" et la lisibilité des APIs fournies aux développeurs et / ou aux clients.

## 2 Gestion des erreurs

Il existe plusieurs solutions, mais un consensus s'est formé autour de l'utilisation des [codes de statut HTTP](#).



Parmi les plus utilisés, on peut citer :

### Succès

Code statut HTTP	Description
200 Ok	Code générique de succès de l'exécution d'un requête, notamment sur les recherches (GET) et les mises à jour (PUT, PATCH)
201 Created	En réponse à la création d'une ressource (POST, PUT)
202 Accepted	Dans un cadre asynchrone, indique que la requête est bien prise en compte pour traitement ultérieur
204 No Content	En réponse à une suppression (DELETE) ou à une recherche (GET) dont les critères ne permettent pas d'avoir une réponse avec contenu.
206 Partial Content	En réponse à une recherche (GET) paginée (cf. HATEOAS)

### Erreurs côté client

Code statut HTTP	Description
400 Bad Request	Code générique face à l'impossibilité de traiter une requête
401 Unauthaurized	lorsque l'utilisateur n'est pas identifié
403 Forbidden	Lorsque l'utilisateur, bien qu'authentifé, ne dispose pas des droits suffisants pour accéder à cette ressource
404 Not Found	La ressource demandée n'existe pas
405 Method Not Allowed	Lorsque la méthode n'est pas applicable à la ressource ou lorsque l'utilisateur n'est pas autorisé à utiliser cette méthode sur la ressource
406 Not Acceptable	incompatibilité de la requête au regard des en-têtes HTTP Accept-*

## Erreurs côté serveur

Code statut HTTP	Description
500 Internal Server Error	Une erreur côté serveur que le client ne peut pas traiter

L'utilisation du code statut HTTP, si elle ne suffit pas, peut être compléter par une structure d'information sur l'erreur en question incluse dans le corps de la réponse (cf. pour exemple la spéc. OAuth2 : [RFC 6749](#)) :

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "error": {
      "type": "string"
    },
    "errorDescription": {
      "type": "string"
    },
    "errorUri": {
      "type": "string"
    }
  },
  "required": [
    "error"
  ]
}
```

## 3 Pour une approche plus fine des requêtes de recherche (GET)

Pour les recherches, il faut utiliser la partie *query string* de l'URI. Ici aussi, il convient d'utiliser la règle définie sur l'ensemble de l'API pour être homogène et cohérent sur l'ensemble de ladite API.

Lorsque la recherche est faite sur une collection ou sur une instance d'une collection, on parlera de **filtre**. Elle sera sous la forme :

```
countries/search?group=europe
```

Si la recherche est multi-ressources (ou globale), on parle de recherche multi-critères. Elle sera sous la forme :

```
/search?q=country+germany
```

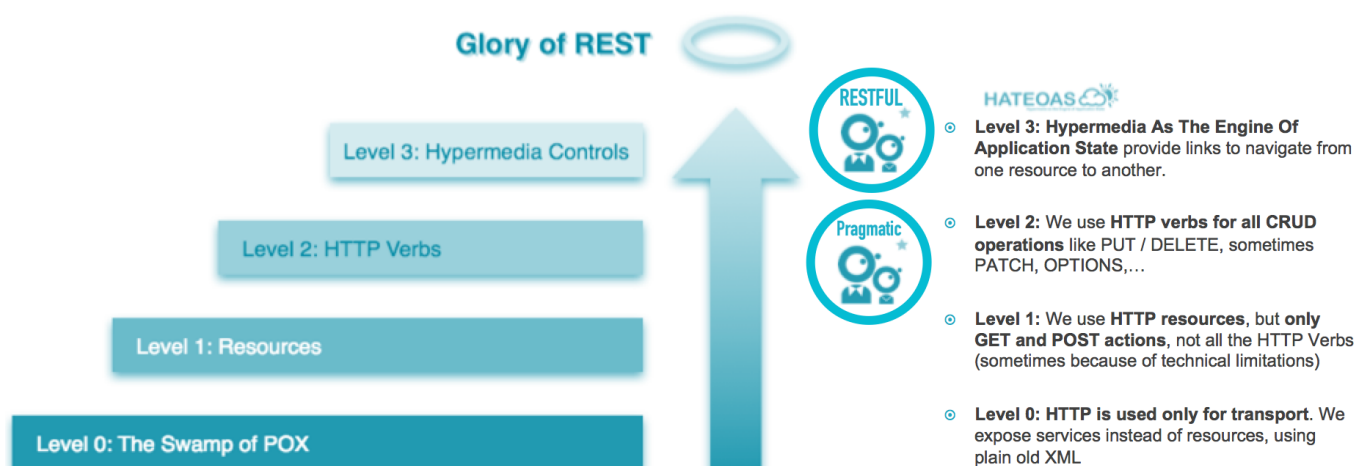
Par ailleurs, les résultats des recherches devraient, le cas échéant, être paginés. L'introduction de la pagination va de facto introduire des mots clés réservés qui ne seront plus utilisables par ailleurs dans l'URI, en général :

- **first**,
- **last**,
- **count**,
- **sort**,
- **desc**

Il faudra aussi adapter l'utilisation des codes statut HTTP avec le code **206 Partial Content**

## 4 HATEOAS ou le Saint Graal REST

Leonard Richardson a développé un modèle de maturité dans l'adoption du style d'architecture REST ([QCon 2008](#)) :



Hypermedia As The Engine Of Application State (HATEOAS) tend à adresser les parts hypermédia et autodescriptive dans la mise en place d'une API.

## 5 Et plus généralement...

**keep it simple, stupid :**



- N'importe quel développeur devrait pouvoir utiliser une API sans être obligé de se référer à la documentation. Mais si l'API est bien documentée, c'est mieux !
- Une API doit être conçue pour les applications clientes, elle peut donc dénormaliser le modèle de données auquel elle se rapporte.
- Les opérations de manipulation des ressources sont uniques
- L'identification des ressources doit utiliser des termes concrets et partagés par tous les acteurs (cf. *ubiquitous language* dans [Domain-Driven Design: Tackling Complexity in the Heart of Software](#), Eric Evans, 2003)
- Développer d'abord les cas d'utilisations *passants*, mais sans oublier qu'il faudra adresser les cas d'utilisations *non passants*.