Le Blog des Octos

Designer une API REST

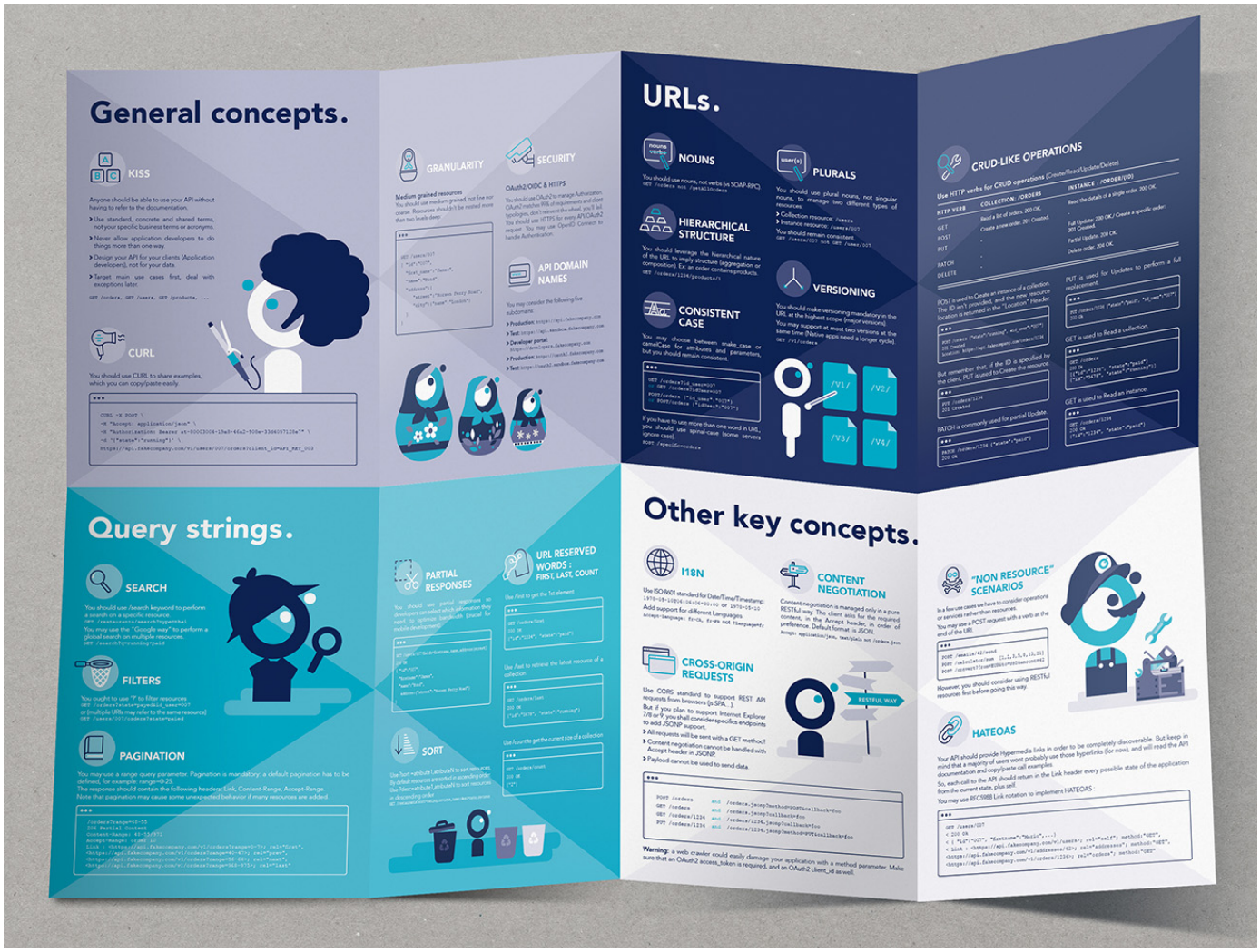
Posté le 01/12/2014 par *Antoine Chantalou*

La période de fêtes approchant à grands pas, nous vous proposons une *"Quick Reference Card"* sur le design des API dont l'objectif est de synthétiser les bonnes pratiques de conception et de design d'API REST.
➡ *Télécharger l'API Design – Quick Reference Card*

➡ *« Vous aimez les API, le Web ? » : Rejoignez nous!*

Si vous avez plus de temps, le présent article reprend – point par point – les éléments de la « carte de référence », en étayant et justifiant les propositions.

Bonne lecture!



  - Réponses partielles

- Query strings
  - Pagination
  - Filtres
  - Tris
  - Recherche
  - Rechercher des ressources
  - Recherche globale
  - Mots réservés : count, last, first, sort…

- Autres concepts clés
  - Négociation de contenu
  - Cross-domain
  - CORS
  - Jsonp
  - HATEOAS

Introduction

Lorsque l'on souhaite concevoir une API, on est rapidement confronté à la problématique du *« design d'API »*. Ce point constitue un enjeu majeur, dans la mesure où une API mal conçue ne sera vraisemblablement peu ou pas utilisée par nos clients : *les développeurs d'applications*.

La mise en oeuvre d'une API à l'*état de l'Art* nécessite de prendre en compte:

- non seulement les principes substantiels des API RESTful issus de la littérature de référence (Roy Fielding, Leonard Richardson, Martin Fowler, spécifications HTTP...)
- mais également les bonnes pratiques utilisées par les API des *"Géants du Web"*.

En effet, il arrive que deux approches s'opposent : celle des *"puristes"*, qui militent pour défendre les principes RESTful sans concession, et celle des *"pragmatiques"* qui privilégient une approche plus pratique, pour que leur API soit fonctionnelle entre les mains d'utilisateurs réels. Bien souvent, la juste réponse se situe au milieu.

Par ailleurs, la phase de conception/design d'une API REST soulève un ensemble de problématiques pour lesquelles les réponses ne sont pas encore unanimes. Les bonnes pratiques REST sont toujours en voie de consolidation et rendent la démarche passionnante.

Afin de faciliter et d'accélérer la mise en oeuvre des API, nous proposons nos convictions, issues de nos expériences autour du sujet API

DISCLAIMER : Cet article constitue un recueil de bonnes pratiques, qui peuvent bien entendu être discutées. Nous vous invitons à participer à la réflexion et à *challenger* ces choix sur notre blog

Concepts généraux

KISS – « Keep it simple, stupid »

Un des objectifs d'une stratégie API vise à "s'ouvrir" sur le WWW, pour toucher un public de développeurs le plus large possible. Il est donc primordial que l'API soit la plus simple possible et auto-descriptive, de manière à ce que l'utilisateur ait à consulter la documentation le moins possible. On parle alors d'affordance : c'est à dire de la capacité de l'API à suggérer son utilisation.

Lors du design d'API, il convient de garder à l'esprit les principes suivants :

- La sémantique d'une API doit être intuitive, qu'il s'agisse de *l'URI*, du *payload* de la *requête* ou *des données retournées* : un utilisateur devrait pouvoir exploiter l'API en ayant recours le moins possible à la documentation de l'API.
  - Les termes utilisés doivent être usuels et concrets : *clients, orders, addresses, products,...* et non tirés d'un *"jargon fonctionnel"*.
  - Il convient de ne pas proposer aux utilisateurs de pouvoir faire quelque chose de plusieurs manières différentes.
  - ➡ Quel "Blender" vous parait être le plus simple d'utilisation

- L'API est "designée" pour les clients : les développeurs, et non pour les "data" (représentation interne des données de l'entreprise). L'API exposée doit exposer des fonctionnalités simples qui répondent aux besoin du client. Une erreur fréquemment rencontrée est de designer son API à partir d'un modèle de données existant, qui est souvent complexe.
  - ➡ Quel "Blender" vous parait être le plus simple d'utilisation

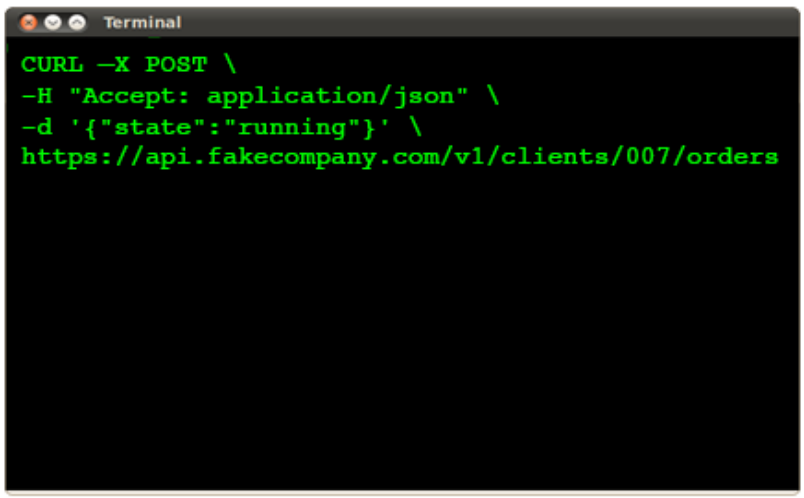
- Enfin, en phase de conception, il convient de se focaliser en premier lieu sur les *"uses-cases"* principaux, et traiter les cas exceptionnels dans un second temps.

Exemples cURL

Chez les *Géants du Web,* les exemples c*URL* sont largement utilisés pour illustrer les appels vers leur API :

- https://developer.github.com/v3/
- https://developers.google.com/youtube/v3/live/authentication#client-side-apps
- https://developer.paypal.com/docs/api/
- https://developers.facebook.com/docs/graph-api/making-multiple-requests
- https://www.dropbox.com/developers/blog/45/using-oauth-20-with-the-core-api
- http://instagram.com/developer/endpoints/likes/
- http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AESDG-chapter-instancedata.html
- ...

Nous préconisons d'illustrer systématiquement vos appels d'API via des exemples cURL dans la documentation de l'API, qui peuvent être utilisés en *copier-coller,* pour lever toute ambiguïté concernant les modalités d'appel.



➡ Exemple

```
CURL –X POST \
-H "Accept: application/json" \
-d '{"state":"running"}' \
https://api.fakecompany.com/v1/clients/007/orders
```

Granularité Moyenne

Si la théorie "une ressource = une URL" pousse à découper l'ensemble des ressources, il nous semble important de garder une limite *raisonnable* dans ce découpage.

Exemple : les informations d'une personne contiennent une adresse courante qui elle-même contient un pays.

Il s'agit d'éviter d'avoir 3 appels à réaliser :

```
CURL https://api.fakecompany.com/v1/users/1234
< 200 OK
< {"id":"1234", "name":"Antoine Jaby", "address":"https://api.fakecompany.com/v1/addresses/4567"}

CURL https://api.fakecompany.com/addresses/4567
< 200 OK
< {"id":"4567", "street":"sunset bd", "country": "http://api.fakecompany.com/v1/countries/98"}

CURL https://api.fakecompany.com/v1/countries/98
< 200 OK
< {"id":"98", "name":"France"}
```

Alors que ces informations sont généralement utilisées ensemble. Ceci peut engendrer des problèmes de performance lié à un trop grand nombre d'appels.

Inversement, si on agrège trop de données a priori, on surcharge inutilement les appels, avec le risque de générer des échanges trop verbeux.

D'une manière générale, exposer une API avec une granularité optimale est souvent culturel, et le fruit de l'expérience sur les problématiques de design d'API. Dans le doute, il faut éviter de cibler ni trop *gros,* ni trop *fin*.

Plus concrètement, nous préconisons :

- De ne grouper que les ressources qui seront accédées à la suite de manière quasi systématique.
- De ne pas grouper les collections pouvant avoir de nombreux composants. Par exemple, la liste des emplois courants sont limités (une personne ne peut pas cumuler beaucoup plus que 2 ou 3 emplois à temps partiel), par contre, la liste des expériences professionnelles peut être très longue.
- De se limiter à deux niveaux d'imbrication :
  - /v1/users/addresses/countries

Noms de domaines des API

Chez les *Géants du Web,* on constate que les domaines sont disparates. Certains utilisent plusieurs domaines ou sous-domaines pour leurs API : c'est notamment le cas de *Dropbox*.

➡ Chez les Géants du Web

API	Domaines / Sous domaines	Exemples d'URI
Google	https://accounts.google.com https://www.googleapis.com https://developers.google.com	https://accounts.google.com/o/oauth2/auth https://www.googleapis.com/oauth2/v1/tokeninfo https://www.googleapis.com/calendar/v3/ https://www.googleapis.com/drive/v2 https://maps.googleapis.com/maps/api/js?v=3.exp https://www.googleapis.com/plus/v1/ https://www.googleapis.com/youtube/v3/ https://developers.google.com
Facebook	https://www.facebook.com https://graph.facebook.com https://developers.facebook.com	https://www.facebook.com/dialog/oauth https://graph.facebook.com/me https://graph.facebook.com/v2.0/{achievement-id} https://graph.facebook.com/v2.0/{comment-id} https://graph.facebook.com/act_{ad_account_id} /adgroups https://developers.facebook.com
Twitter	https://api.twitter.com https://stream.twitter.com https://dev.twitter.com	https://api.twitter.com/oauth/authorize https://api.twitter.com/1.1/statuses/show.json https://stream.twitter.com/1.1/statuses/sample.json https://dev.twitter.com
GitHub	https://github.com https://api.github.com https://developer.github.com	https://github.com/login/oauth/authorize https://api.github.com/repos/octocat/Hello-World/git /commits /7638417db6d59f3c431d3e1f261cc637155684cd https://developer.github.com
Dropbox	https://www.dropbox.com https://api.dropbox.com https://api-content.dropbox.com https://api-notify.dropbox.com	https://www.dropbox.com/1/oauth2/authorize https://api.dropbox.com/1/account/info https://api-content.dropbox.com/1/files/auto/ https://api-notify.dropbox.com/1/longpoll_delta https://api-content.dropbox.com/1/thumbnails/auto/ https://www.dropbox.com/developers

Instagram	https://api.instagram.com http://instagram.com	https://api.instagram.com/oauth/authorize/ https://api.instagram.com/v1/media/popular http://instagram.com/developer/
Foursquare	https://foursquare.com https://api.foursquare.com https://developer.foursquare.com	https://foursquare.com/oauth2/authenticate https://api.foursquare.com/v2/venues /40a55d80f964a52020f31ee3 https://developer.foursquare.com

Afin de normaliser les noms de domaines, dans un souci d'affordance, nous préconisons d'utiliser uniquement trois sous-domaines pour la production :

- API – https://api.{fakecompany}.com
- OAuth2 – https://oauth2.{fakecompany}.com
- Portal developer – https://developers.{fakecompany}.com

L'objectif est que le développeur, qui consomme l'API, puisse déterminer de manière intuitive quel domaine appeler, en fonction qu'il souhaite :

1. Appeler l'API
2. Récupérer un jeton pour appeler l'API (via OAuth2)
3. Consulter le "portail developer" de l'API

Par ailleurs, *Paypal* propose un environnement "*sandbox*" pour son API, très pratique pour pouvoir réaliser des tests "*Try-It*" : https://developer.paypal.com/docs/api/

Nous proposons d'utiliser deux sous-domaines spécifiques pour une plate-forme de tests :

- OAuth2 – https://oauth2.sandbox.{fakecompany}.com
- API – https://api.sandbox.{fakecompany}.com

Sécurité

Deux protocoles sont généralement utilisés pour sécuriser les API REST :

- OAuth1 : http://tools.ietf.org/html/rfc5849
- OAuth2 : http://tools.ietf.org/html/rfc6749

➡ Chez les Géants du Web et dans les DSI

OAuth1	OAuth2
Twitter, Yahoo, flickr, tumblr, Netflix, myspace, evernote,...	*Google, Facebook, Dropbox, GitHub, amazone, Intagram, LinkedIn, foursquare, salesforce, viadeo, Deezer, Paypal, Stripe, huddle, boc, Basecamp, bitly,...*
MasterCard, CA-Store, OpenBankProject, intuit,...	*AXA Banque, Bouygues telecom,...*

Nous préconisons de sécuriser votre API via le protocole *OAuth2*.

- Contrairement à OAuth1, OAuth2 permet de gérer l'authentification et l'habilitation des ressources par tout type d'application (native mobile, native tablette, application javascript, application web de type serveur, application batch/back-office, ...) avec ou sans consentement de l'utilisateur propriétaire des ressources.

- OAuth2 est le standard de sécurisation des API : proposer un protocole marginal freinerait vraisemblablement l'adoption de votre API.
- D'autre part, les problématiques de sécurisation des ressources sont complexes, et une solution propriétaire entraînerait à coût sur des risques non négligeables.

Nous proposons d'implémenter la préconisation de *Google* pour le f*low OAuth2 implicit* relative à la validation du token :

- https://developers.google.com/accounts/docs/OAuth2UserAgent#validatetoken
- http://en.wikipedia.org/wiki/Confused_deputy_problem

Nous préconisons d'utiliser systématiquement le protocole HTTPS pour tous les accès vers :

- les providers OAuth2
- les providers d'API

Un excellent test pour valider la bonne implémentation ou mise en oeuvre de votre solution OAuth2 consiste à réaliser un appel avec le même code client sur votre API, et sur l'API Google par exemple, en ayant à changer uniquement les noms de domaine des serveurs OAuth2 et API.

URIs

Noms > verbes

Pour décrire vos ressources, nous préconisons d'utiliser des noms concrets, pas de verbe.

En informatique, nous utilisons depuis longtemps les verbes pour exposer des services avec une approche RPC, par exemple :

- getClient(1)
- creerClient()
- majSoldeCompte(1)
- ajoutetProduitDansCommande(1)
- effacerAdresse(1)
- ...

Mais dans une approche RESTful, nous préconisons d'utiliser :

- GET /clients/1
- POST /clients

- PATCH /accounts/1
- PUT /orders/1
- DELETE /addresses/1
- ...

Un des objectifs fondamentaux d'une API REST est précisément d'utiliser HTTP comme protocole applicatif, pour homogénéiser et faciliter les interactions entre SI et pour ne pas avoir à façonner un protocole « maison » de type *SOAP/RPC* ou *EJB,* qui a l'inconvénient de "réinventer la roue" à chaque fois.

Il convient donc d'utiliser systématiquement les verbes HTTP pour décrire les actions réalisées sur les ressources (voir rubrique CRUD).

En outre, l'utilisation des verbes HTTP rend l'API intuitive et permet d'éviter que le développeur n'ait à consulter une documentation verbeuse pour comprendre comment manipuler les ressources, favorisant ainsi l'affordance de l'API.

En pratique, le développeur client est en général équipé d'un outillage (bibliothèques et framework) qui permet de générer des requêtes HTTP avec les verbes adéquates, à partir d'un modèle objet, lorsque ce dernier est mis à jour.

Pluriel > singulier

La plupart du temps, les *Géants du Web* restent cohérents entre les noms de ressource au singulier et les noms au pluriel. L'enjeu principal est effectivement de ne pas les mélanger. En effet, varier les noms de ressources entre pluriel et singulier freine "l'explorabilité" de l'API.

Par ailleurs, les noms de ressources nous semblent être plus naturel au pluriel afin d'adresser de manière cohérente les *collections* et *instances* de ressources.

Nous préconisons donc d'utiliser le pluriel pour gérer les deux type de ressources :

- Collection de ressources : /v1/users
- Instance d'une ressource : /v1/users/007

Pour la création d'un utilisateur par exemple, on considère que *POST /v1/users* est l'invocation de l'action de création sur la collection des *users*. De même, pour récupérer un utilisateur, *GET /v1/users/007*, se lit comme "Je veux l'utilisateur 007 dans cette collection d'utilisateurs".

Casse cohérente

Casse des URI

Lorsqu'il s'agit de nommer des ressources dans un programme informatique, on remarque majoritairement 3 types de style : *CamelCase*, *snake_case* et *spinal-case*. Il s'agit de nommer des ressources de manière proche du langage naturel en évitant toutefois d'utiliser des espaces, des apostrophes ou des caractères jugés trop exotiques. Cette pratique s'est imposée dans les langages de programmation où un ensemble réduit de caractères est autorisé pour identifier les variables.

- CamelCase : Cette méthode a été démocratisée par le langage Java. Il s'agit de démarquer le début de chaque mot en mettant la première lettre en majuscule. Par ex. *CamelCase, CurrentUser, AddAttributeToGroup,* etc. En dehors des débats d'experts sur la lisibilité, son principal défaut est d'être inutilisable dans les contextes insensibles à la casse. Il existe deux variantes :
  - lowerCamelCase : où la première lettre est en minuscule.
  - UpperCamelCase : où la première lettre est en majuscule.

- snake_case : Cette méthode est celle utilisée depuis longtemps en C et plus récemment en Ruby. Les mots sont alors séparés par des underscore « _ » permettant à un compilateur ou un interprêteur de le comprendre en tant que symbole unique, mais permettant au lecteur humain de séparer les mots de manière quasi naturelle. Sa baisse de popularité est en partie due aux abus des programmes en C, utilisant soit des noms à rallonge soit des noms ultra abrégés. À l'inverse du camel case, il existe très peu de contextes dans lesquels il est incompatible. Quelques exemples : *snake_case, current_user, add_attribute_to_group*, etc.
- spinal-case : Variante du snake case, le spinal case utilise des tirets courts « – » pour séparer les mots. Les avantages et inconvénients sont en grandes partie identiques à snake case, à l'exception que de nombreux langages de programmation ne peuvent l'accepter en tant que symbole (nom de variable ou fonction). Il est parfois appelé lisp-case car en dialectes LISP, c'est la manière habituelle de nommer les variables et les fonctions. C'est

> aussi traditionnellement la manière dont les dossiers et les fichiers sont nommés dans les systèmes UNIX et Linux. Exemples : *spinal-case, current-user, add-attribute-to-group*, etc.

Ces trois types de nommages ont chacun leurs propres variantes en fonction d'autres critères comme la casse utilisée pour la première lettre ou le traitement réservé aux accents et autres caractères spéciaux. De manière générale, on préfère de toute façon utiliser l'anglais basique, exempt de caractères spéciaux.

D'après la RFC3986, les url sont "case sensitive" (hormis le *scheme* et le *host*). Mais en pratique, une casse sensitive peut entraîner des dysfonctionnements pour les API hébergées sous *Windows*.

Chez les Géants du Web :

	Google	Facebook	Twitter	Paypal	Amazon	dropbox	github
snake_case	x	x	x			x	x
spinal-case	x			x			
camelCase	x						

Nous recommandons d'utiliser pour les URI une *casse cohérente*, à choisir entre :

- spinal-case (mise en avant par la RFC3986)
- et snake_case (fréquemment utilisée par les *Géants du Web*)

➡ Exemples

```
POST /v1/specific-orders
```

ou

```
POST /v1/specific_orders
```

Casse du body

Pour les données contenues dans le body, deux formats sont majoritairement utilisés.

La notation snake_case est sensiblement plus utilisée par les *Géants du Web* et notament adoptée par les spécifications *OAuth2*. En revanche, la popularité croissante du langage *Javascript* contribue significativement à l'adoption de la notation camelCase, même si sur le papier, REST devrait prôner l'indépendance du language et permettre d'exposer une API à

l'état de l'art sur *Xml*.

Nous recommandons d'utiliser pour les URI une casse cohérente, à choisir entre :

- snake_case (fréquemment utilisée la communauté Web Ruby...)
- lowerCamlCase (fréquemment utilisée par les communautés Javascript, Java...)

➡ Exemples

```
GET  /orders?id_client=007        or  GET /orders?idClient=007
POST /orders {"id_client":"007"}  or  POST/orders {"idClient":"007”}
```

Versioning

Toute API sera amenée a évoluer dans le temps. Une API peut être versionée de différentes manières:

- Par *timestamp,* par *numero de version,...*
- Dans le *path,* au début ou à la fin de l'*URI*
- En paramètre de la *request*
- Dans un *Header HTTP*
- Avec un versioning *facultatif* ou *obligatoire*

➡ Chez les Géants du Web

API	Versioning
Google	*URI path or parameter* https://www.googleapis.com/oauth2/v1/tokeninfo https://www.googleapis.com/calendar/v3/ https://www.googleapis.com/drive/v2 https://maps.googleapis.com/maps/api/js?v=3.exp https://www.googleapis.com/plus/v1/ https://www.googleapis.com/youtube/v3/
Facebook	*URI (optional)* https://graph.facebook.com/v2.0/{achievement-id} https://graph.facebook.com/v2.0/{comment-id}
Twitter	https://api.twitter.com/1.1/statuses/show.json https://stream.twitter.com/1.1/statuses/sample.json

API	Versioning
GitHub	*Accept Header (optional)* Accept: application/vnd.github.v3+json
Dropbox	*URI* https://www.dropbox.com/1/oauth2/authorize https://api.dropbox.com/1/account/info https://api-content.dropbox.com/1/files/auto https://api-notify.dropbox.com/1/longpoll_delta https://api-content.dropbox.com/1/thumbnails/auto
Instagram	*URI* https://api.instagram.com/v1/media/popular
Foursquare	*URI* https://api.foursquare.com/v2/venues/40a55d80f964a52020f31ee3
LinkedIn	*URI* http://api.linkedin.com/v1/people/
Netflix	*URI, optionel parameter* http://api.netflix.com/catalog/titles/series/70023522?v=1.5
Paypal	*URI* https://api.sandbox.paypal.com/v1/payments/payment

Nous préconisons de faire figurer un numéro de version obligatoire, sur un digit, au plus haut niveau du *path* de *l'uri*.

- Le numéro désigne une version majeure de l'API pour une ressource donnée, qui impose un changement pour que l'API fonctionne
- REST et Json permettent, entre autre, par rapport à SOAP/xml, une certaine flexibilité pour faire évoluer l'API sans avoir à impacter (redéployer) les clients. Par exemple en ajoutant des attributs à une ressource existante. La version de l'API n'a pas à être incrémentée dans ce cas là.
- Le versionning par défaut est à proscrire car en cas de changement de l'API, les impacts sur les applications appelantes seraient non maîtrisés par les clients.
- La version d'une API étant une information essentielle, nous privilégions, pour des problématiques d'affordance, de le faire apparaître dans l'URL plutôt que dans le header HTTP.
- Nous préconisons de supporter au plus, deux versions en parallèle (le cycle d'adoption par les applications *natives* est souvent plus long).

➡ Exemple

```
GET /v1/orders
```

CRUD

Comme nous l'avons indiqué, un des objectifs fondamentaux de l'approche REST est précisément d'utiliser HTTP comme protocole applicatif pour ne pas avoir à façonner une API « maison ».

Il convient donc d'utiliser systématiquement les verbes HTTP pour décrire les actions réalisées sur les ressources, et de faciliter le travail du développeur pour les manipulations CRUD récurrentes. Le tableau suivant synthétise les bonnes pratiques généralement constatées :

Verbe HTTP	Correspondance CRUD	Collection : /orders	Instance : /orders/{id}
GET	READ	Read a list orders. 200 OK.	Read the detail of a single order. 200 OK.
POST	CREATE	Create a new order. 201 Created.	–
PUT	UPDATE/CREATE	–	Full Update. 200 OK. Create a specific order. 201 Created.
PATCH	UPDATE	–	Partial Update. 200 OK.
DELETE	DELETE	–	Delete order. 200 OK.

Le verbe HTTP POST est utilisé pour créer une instance au sein d'une collection. L'identifiant de la ressource à créer ne doit pas être précisé :

```
CURL –X POST \
-H "Accept: application/json" \
-H "Content-Type: application/json" \
-d '{"state":"running","id_client":"007"}' \
https://api.fakecompany.com/v1/clients/007/orders

< 201 Created
< Location: https://api.fakecompany.com/v1/clients/007/orders/1234
```

Le code retour n'est pas 200 mais 201.
L'URI et l'identifiant de la nouvelle ressource sont retournés dans le *header* "*Location*" de la réponse.

Si l'identifiant de la ressource est spécifié par le client, le verbe HTTP PUT est utilisé pour la création d'une instance de la collection. Cependant, en pratique, ce cas d'utilisation est moins fréquent.

```
CURL –X PUT \
-H "Content-Type: application/json" \
-d '{"state":"running","id_client":"007"}' \
https://api.fakecompany.com/v1/clients/007/orders/1234
< 201 Created
```

Le verbe HTTP PUT est en utilisé systématiquement pour réaliser une mise à jour totale d'une instance de la collection (tous les attributs sont remplacés et ceux qui sont non présents seront supprimés).

Dans l'exemple ci-dessous, on met à jour l'attribut *state* et l'attribut *id_client*. Tous les autres champs seront supprimés.

```
CURL –X PUT \
-H "Content-Type: application/json" \
-d '{"state":"paid","id_client":"007"}' \
https://api.fakecompany.com/v1/clients/007/orders/1234

< 200 OK
```

Le verbe HTTP PATCH (non présent initialement dans les spécifications HTTP mais ajouté ultérieurement) est couramment utilisé pour une mise à jour partielle d'une instance de la collection.

Dans l'exemple ci-dessous, on met à jour l'attribut *state*, mais les autres attributs restent tel quel.

```
CURL –X PATCH \
-H "Content-Type: application/json" \
-d '{"state":"paid"}' \
https://api.fakecompany.com/v1/clients/007/orders/1234

< 200 OK
```

Le verbe HTTP GET est utilisé pour lire une collection. En pratique, l'API ne retourne généralement pas l'intégralité des réponses (voir section Pagination).

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/clients/007/orders

< 200 OK
< [{"id":"1234", "state":"paid"}, {"id":"5678", "state":"running"}]
```

Le verbe HTTP GET est utilisé pour lire l'instance d'une collection.

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/clients/007/orders/1234

< 200 OK
< {"id":"1234", "state":"paid"}
```

Réponses partielles

Les réponses partielles permettent au client de récupérer uniquement les informations dont il a besoin. Cette fonctionnalité est par ailleurs primordiale pour les contextes en utilisation nomade (3G-), ou la bande passante doit être optimisée.

➡ Chez les Géants du Web

API	Partial responses
Google	?fields=url,object(content,attachments/url)
Facebook	&fields=likes,checkins,products
LinkedIn	https://api.linkedin.com/v1/people/~:(id,first-name,last-name,industry)

A minima, nous préconisons de pouvoir sélectionner les attributs à remonter, sur un niveau de ressource, via la notation Google *fields=attribute1,attributeN* :

```
GET /clients/007?fields=firstname,name
200 OK
{
  "id":"007",
  "firstname":"James",
  "name":"Bond"
}
```

Pour les cas où les performances constituent un enjeu fort, nous proposons, d'utiliser la notation Google *fields=object(attribute1,attributeN)*. Par exemple pour ne récupérer que le *prénom*, le *nom* et la *rue* de *l'adresse* d'un client :

```
GET /clients/007?fields=firstname,name,address(street)
200 OK
{
  "id":"007",
  "firstname":"James",
  "name":"Bond",
  "address":{"street":"Horsen Ferry Road"}
}
```

Query strings

Pagination

Il est nécessaire de prévoir dès le début de votre API la *pagination* de vos *ressources*. En effet, il est difficile d'anticiper avec exactitude l'évolution de la quantité de données qui sera retournée. C'est pourquoi nous recommandons de *paginer* vos ressources avec des valeurs par défaut lorsque celles-ci ne sont pas spécifiées par l'appelant, par exemple avec une plage de valeurs [0-25].

La pagination systématique apporte aussi une homogénéité de vos ressources, ce qui ne peut être que bénéfique car n'oublions pas que la description d'une API doit être autoportante : moins vos consommateurs auront de documentations à lire, plus ils s'approprieront votre API.

➡ Chez les Géants du Web

API	Pagination
Facebook	Paramètres : before, after, limit, next, previews

```
"paging": {
  "cursors": {
    "after": "MTAxNTExOTQ1MjAwNzI5NDE=",
    "before": "NDMyNzQyODI3OTQw"
  },
  "previous": "https://graph.facebook.com/me/albums?limit=25&before=NDMyNzQyODI3OTQw"
  "next": "https://graph.facebook.com/me/albums?limit=25&after=MTAxNTExOTQ1MjAwNzI5NDE="
}
```

Google	Paramètres : maxResults, pageToken

```
"nextPageToken":"CiAKGjBpNDd2Nmp2Zml2cXRwYjBpOXA",
```

Twitter Paramètres : since_id, max_id, count


```
"next_results": "?max_id=249279667666817023&q=*freebandnames&count=4&include_entities=1&result_type=mixed",
"count": 4,
"completed_in": 0.035,
"since_id_str": "24012619984051000",
"query": "*freebandnames",
"max_id_str": "250126199840518145"
```


GitHub Paramètres : page, per_page


```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
<https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```


Paypal Paramètres : start_id, count


```
{'count': 1,'next_id': 'PAY-5TU010975T094876HKKDU7MZ',
```


Divers mécanismes de pagination sont utilisés par les *Géants du Web*. Puisqu'aucun principe commun ne se dégage véritablement, nous proposons d'utiliser :

- le paramètre de requête ?range=0-25
- et les *Header* standards *HTTP pour la réponse* :
  - Content-Range
  - Accept-Range

➡ La pagination dans la requête
D'un point de vue pratique, la pagination souvent gérée dans l'url via la *query-string*. Les en-têtes HTTP fournissent également ce mécanisme. Nous proposons d'accepter uniquement la solution via *query-string*, et de ne pas tenir compte du *Header Range HTTP*. La pagination est une information importante qui par souci d'affordance peut être positionnée dans la requête.

Nous vos proposons d'utiliser une plage de valeurs, via l'index des ressources de votre collection. Par exemple, les ressources de l'index 10 à l'index 25 inclus équivaut à ?range=10-25.

➡ La pagination dans la réponse
Le code retour HTTP correspondant au retour d'une requête paginée sera *206 Partial Content*. Sauf, si les valeurs demandées provoquent la remonté de l'ensemble des données de la collections, auquel cas le code retour sera *200 Ok*.

La réponse de votre API sur une collection devra obligatoirement fournir dans les en-têtes HTTP :

- Content-Range *offset – limit / count*
  - *offset* : l'index du premier élément retourné par la requête.
  - *limit* : l'index du dernier élément retourné par la requête.
  - *count* : le nombre total d'élément que contient la collection.

- Accept-Range *resource max*
  - *resource* : le type de la pagination, on parlera ici systématiquement de la ressource en cours d'utilisation, *ex : client, order, restaurant, ...*
  - *max* : le nombre maximum pouvant être requêté en une seule fois.

Dans le cas où la pagination demandée ne rentre pas dans les valeurs tolérées par l'API, la réponse HTTP sera un code erreur 400, avec une description explicite de l'erreur dans le body.

➡ Liens de navigation
Il est fortement conseillé d'incorporer dans les entêtes HTTP de vos réponses la balise *Link*. Celle-ci vous permet d'ajouter, entre autre, des liens de navigations tel que la page suivante, la page précédente, la première page, la dernière page etc...

➡ Exemples
Nous avons dans notre API une collection de 48 restaurants, pour laquelle il n'est possible d'en consulter que 50 par requête. La pagination par défaut est 0-50 :

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants
< 200 Ok
< Content-Range: 0-47/48
< Accept-Range: restaurant 50
< [...]
```

Si 25 ressources sont demandées, sur les 48 disponibles, un code *206 Partial Content* est retourné :

```
CURL -X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?range=0-24
< 206 Partial Content
< Content-Range: 0-24/48
< Accept-Range: restaurant 50
```

Si 50 ressources sont demandées, sur les 48 disponibles, un code *200 OK* est retourné :

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?range=0-50
< 200 Ok
< Content-Range: 0-47/48
< Accept-Range: restaurant 50
```

Si la plage demandée est supérieure au nombre de ressources maximum pouvant être requêté en une seule fois (*Header Accept-Range*), un code *400 KO* est retourné :

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/orders?range=0-50
< 400 Bad Request
< Accept-Range: order 10
< { reason : "Requested range not allowed" }
```

Pour retourner les liens vers les autres plages, nous préconisons la notation ci-dessous, utilisée par GitHub, compatible avec la RFC5988 (et qui permet de gérer les clients qui ne supportent pas plusieurs *Header "Link"*)

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/orders?range=48-55
< 206 Partial Content
< Content-Range: 48-55/971
< Accept-Range: order 10
< Link : <https://api.fakecompany.com/v1/orders?range=0-7>; rel="first", <https://api.fakecompany.com/v1/
orders?range=40-47>; rel="prev", <https://api.fakecompany.com/v1/orders?range=56-64>; rel="next", <htt
ps://api.fakecompany.com/v1/orders?range=968-975>; rel="last"
```

Une autre notation est fréquemment rencontrée, où la balise d'en-tête *HTTP Link* est constituée d'une *URL* suivi du type de liens associés. Cette balise est répétable autant de fois qu'il y a de liens associés à votre réponse :

```
< Link: <https://api.fakecompany.com/v1/orders?range=0-7>; rel="first"
< Link: <https://api.fakecompany.com/v1/orders?range=40-47>; rel="prev"
< Link: <https://api.fakecompany.com/v1/orders?range=56-64>; rel="next"
< Link: <https://api.fakecompany.com/v1/orders?range=968-975>; rel="last"
```

Ou bien la notation suivante dans le payload, utilisée par Paypal :

```
[
  {"href":"https://api.fakecompany.com/v1/orders?range=0-7", "rel":"first", "method":"GET"},
  {"href":"https://api.fakecompany.com/v1/orders?range=40-47", "rel":"prev", "method":"GET"},
  {"href":"https://api.fakecompany.com/v1/orders?range=56-64", "rel":"next", "method":"GET"},
  {"href":"https://api.fakecompany.com/v1/orders?range=968-975", "rel":"last", "method":"GET"},
]
```

Filtres

Un *filtre* consiste à limiter le nombre de ressources requêtées, en spécifiant des attributs et leurs valeurs correspondantes attendues. Il est possible de filtrer une collection sur plusieurs attributs simultanément, et de permettre plusieurs valeurs pour un même attribut filtré.

Pour cela, nous proposons d'utiliser directement le nom de l'attribut avec une égalité sur les valeurs attendues, chacune séparées par une virgule.

Exemple : récupération des restaurants faisant de la cuisine *thai*

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?type=thai
```

Exemple: récupération des restaurants avec un *rating* de 4 ou 5, offrant de la cuisine *chinese* ou *japanese* , ouvert le *sunday*

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?type=japanese,chinese&rating=4,5&days=sunday
```

Tris

Le tri du résultat d'un appel sur une collection de ressources passe par deux principaux paramètres :

- *sort* : contient les noms des attributs, séparés par une virgule, sur lesquels effectuer le trie.
- *desc* : par défaut le tri est ascendant (ou croissant), afin de l'obtenir de façon descendant (ou décroissant), il suffit d'ajouter ce paramètre (sans valeur par défaut). On voudra dans certains cas spécifier quels attributs doivent être traités de façon ascendant ou descendant, on mettra alors dans ce paramètre la liste des attributs descendants, les autres seront donc par défaut ascendants.

Exemple : récupération de la liste des restaurants triée alphabétiquement sur le nom.

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?sort=name
```

Exemple : récupération de la liste des restaurants, triée par rating décroissant, puis par nombre de reviews décroissant, et enfin alphabétiquement sur le name.

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?sort=rating,reviews,name&desc=rating,reviews
```

➡ Tri, Filtre et Pagination
La pagination sera nécessairement impactée par les changements de trie et de filtre. La combinaison des trois compléments de requête doit pouvoir être imbriquée en toute cohérence dans vos requêtes d'API.

Exemple : requête des 5 premiers restaurants chinese trié par rating descendant.

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants?type=chinese&sort=rating,name&desc=rating&range=0-4
< 206 Partial Content
< Content-Range: 0-4/12
< Accept-Range: restaurants 50
```

Recherche

Rechercher des ressources

Lorsque que le filtrage ne suffit pas (pour faire du partiel ou de l'approchant par exemple), on passera alors par une recherche sur les ressources.

Une recherche est en elle-même une sous-ressource, de votre collection, car les résultats qu'elle fournit auront un format différent des ressources recherchées et de la collection elle-même. Cela permet d'ajouter des suggestions, des corrections et des infos propres à la recherche.

Les paramètres sont fournis de la même manière que pour le filtre, via la query-string, mais ceux-ci ne seront pas nécessairement des valeurs exacts, et pourront avoir une nomenclature permettant de faire de l'approchant.

La recherche étant une ressource à part entière, elle doit supporter la pagination de la même manière que les autres ressources de votre API.

Exemple : *recherche des restaurants dont le name commence par « La »*.

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/restaurants/search?name=la*
< 206 Partial Content
< { "count" : 5, "query" : "name=la*", "suggestions" : ["las"], results : [...] }
```

Exemple : *recherche des 10 premiers restaurants ayant « napoli » dans leur name, faisant de la cuisine chinese ou japanese, situé dans le 75 (Paris), trié selon leur rating descendant et leur name alphabétiquement.*

```
CURL –X GET \
-H "Accept: application/json" \
-H "Range 0-9" \
https://api.fakecompany.com/v1/restaurants/search?name=*napoli*&type=chinese,japanese&zipcode=75*&sort=rating,name&desc=rating&range=0-9
< 206 Partial Content
< Content-Range: 0-9/18
< Accept-Range: search 20
< { "count" : 18, "range": "0-9", "query" : "name=*napoli*&type=chinese,japanese&zipcode=75*", "suggestions" : ["napolitano", "napolitain"], results : [...] }
```

Recherche globale

La recherche globale se comportera de la même manière qu´une recherche par ressource, celle-ci sera simplement située à la racine de votre API et devra être spécifiée clairement dans votre documentation.

Nous préconisons la notation utilisée par Google pour les recherches globales :

```
CURL –X GET \
-H "Accept: application/json" \
https://api.fakecompany.com/v1/search?q=running+paid
< [...]
```

Autres concepts clés

Négociation de contenu

Nous recommandons de gérer plusieurs format de distribution du contenu de votre API. On utilisera pour cela l´entête HTTP prévue à cet effet: « Accept »,
Par défaut, l´API pourra distribuer les ressources au format JSON, mais dans les cas où la requête spécifiera en premier lieu « Accept: application/xml », les ressources seront fournis au format XML.

Il est conseillé de gérer à minima 2 formats: JSON et XML. L´ordre des formats demandés par l´entête « Accept », doit être respecté pour définir le format de la réponse.
Dans les cas où il n´est pas possible de fournir le format demandé, une erreur http 406 est retournée (cf Erreurs – Status Codes).

```
GET https://api.fakecompany.com/v1/offers
Accept: application/xml; application/json      XML préféré à JSON
< 200 OK
< [XML]

GET https://api.fakecompany.com/v1/offers
Accept: text/plain; application/json           Text non pris en charge par l'api
< 200 OK
< [JSON]
```

Cross-domain

CORS

Lorsque l´application (javsacript SPA) et l´API se trouvent sur des domaines différents, par exemple :

- https://fakeapp.com
- https://api.fakecompany.com

une bonne pratique consiste à utiliser le protocole CORS qui est le standard HTTP.

La mise en oeuvre de CORS coté serveur consiste en général à ajouter quelques directives sur les serveurs HTTP (Nginx/Apache/Nodejs...).

Coté client, la mise en oeuvre est transparente : le navigateur effectuera avant chaque requête GET/POST/PUT/PATCH/DELETE une requêtes HTTP avec le verbe OPTIONS.

Voici par exemple les deux appels successifs que réalisera un navigateur pour récupérer, via GET, les informations d´un utilisateur sur l´API Google+ :

```
CURL -X OPTIONS \
-H 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8' \
'https://www.googleapis.com/plus/v1/people/105883339188350220174?client_id=API_KEY'
```

```
CURL -X GET\
'https://www.googleapis.com/plus/v1/people/105883339188350220174?client_id=API_KEY' \
-H 'Accept: application/json, text/javascript, */*; q=0.01'\
-H 'Authorization: Bearer foo_access_token'
```

Jsonp

En pratique, CORS n'est pas (ou très mal) supporté par les anciens navigateurs, notamment IE7,8 et 9. Si votre API est utilisée par des navigateurs que vous ne maîtrisez pas (sur Internet, avec des utilisateurs finaux), il encore nécessaire de proposer une exposition *Jsonp* de API, en *fallback* de la mise en oeuvre *CORS*.

En pratique, Jsonp est un coutournement de l'usage du tag *< script />* visant à permettre la gestion du *cross-domain*. L'utilisation de Jsonp, entraîne certaines limitations :

- Il est impossible d'exploiter la négociation de contenu via le *Header Accept* => Un nouveau *endpoint* doit être publié, avec l'extension .jsonp par exemple, pour permettre au contrôleur de déterminer qu'il s'agit d'une requête jsonp.
- Toutes les requêtes sont émises via le verbe HTTP GET => un paramètre *method=XXX* doit donc être proposé
  - Garder à l'esprit qu'un *web crawler* pourrait endommager lourdement vos données si aucun contrôle d'habilitation n'est réalisé sur l'appel *method=DELETE* par exemple...

- Le *payload* de la requête ne peut être exploité pour acheminer des données => toutes les données doivent être envoyées en paramètre de requête.

Pour être *CORS & Jsonp compliant,* votre API doit par exemple exposer les *endpoints* suivants :

```
POST /orders       et  /orders.jsonp?method=POST&callback=foo
GET  /orders       et  /orders.jsonp?callback=foo
GET  /orders/1234  et  /orders/1234.jsonp?callback=foo
PUT  /orders/1234  et  /orders/1234.jsonp?method=PUT&callback=foo
```

HATEOAS

Concept

Prenons l'exemple *d'Angélina Jolie*, cliente *d'Amazon* qui souhaite consulter les détails de sa dernière commande. Pour cela, elle va devoir réaliser deux actions:

1. Lister toutes ses commandes
2. Sélectionner sa dernière commande

Sur le site web *d'Amazon*, *Angélina* n'a pas besoin d'être experte en *web* pour consulter sa dernière commande : il lui suffit se connecter sur le site avec son compte, de cliquer sur le lien "mes commandes" puis de sélectionner la dernière.

Imaginons qu'*Angélina* souhaite utiliser une API pour effectuer la même action!

Elle doit commencer par consulter la documentation *d'Amazon* pour trouver l'URL qui va lui permettre de lister ses commandes. Une fois trouvée, elle doit jouer l'url qui lui retournera la liste des commandes. Elle verra alors sa commande, mais pour y accéder il lui faudra une autre URL. *Agelina* devra donc consulter la documentation pour construire l'URL adaptée.

La différence entre ces deux scénarios c'est que dans le premier, *Angélina* n'a eu a connaître que la première URL: "http://www.amazon.com" puis à se laisser guider par les liens présents dans la page web. Or dans le deuxième cas, *Angélina* a été contrainte de consulter la documentation pour construire l'URL.

L'inconvénient de la seconde démarche est que :

- En condition réelle, la documentation n'est pas toujours à jour. *Angélina* peut passer à coté d'un ou plusieurs services disponibles parce qu'ils ne sont pas correctement documentés.
- *Angelina* est généralement une développeuse et les développeurs n'affectionnent pas particulièrement la documentation.
- L'API perd en accessibilité.

Supposant que notre *Angelina* développe un batch pour automatiser cette action. Que va t-il se passer lorsque *Amazon* modifiera ses URLs ?

Implémentation

En pratique, HATEOAS c'est un peu comme la météo : "Everybody talks about the weather but nobody does anything about it. »

Chez les Géants du Web, Paypal propose une implémentation :

```
[
  {
    "href": "https://api.sandbox.paypal.com/v1/payments/payment/PAY-6RV70583SB702805EKEYSZ6Y",
    "rel": "self",
    "method": "GET"
  },
  {
    "href": "https://www.sandbox.paypal.com/webscr?cmd=_express-checkout&token=EC-60U79048BN771960
9",
    "rel": "approval_url",
    "method": "REDIRECT"
  },
  {
    "href": "https://api.sandbox.paypal.com/v1/payments/payment/PAY-6RV70583SB702805EKEYSZ6Y/execut
e",
    "rel": "execute",
    "method": "POST"
  }
]
```

En utilisant cette notation, un appel à */clients/007* retournerait les informations *client*, mais également des pointeurs vers les différents états possibles :

```
GET /clients/007
< 200 Ok
< { "id":"007", "firstname":"James",...,
  "links": [
    {"rel":"self","href":"https://api.domain.com/v1/clients/007", "method":"GET"},
    {"rel":"addresses","href":"https://api.domain.com/v1/addresses/42", "method":"GET"},
    {"rel":"orders", "href":"https://api.domain.com/v1/orders/1234", "method":"GET"},
    ...
  ]
}
```

Pour l'implémentation de HATEOAS, nous proposons d'utiliser la notation ci-dessous, utilisée par GitHub, compatible avec la RFC5988 (et qui permet de gérer les client qui ne supportent pas plusieurs *Header "Link"*) :

```
GET /clients/007
< 200 Ok
< { "id":"007", "firstname":"James",...}
< Link : <https://api.fakecompany.com/v1/clients>; rel="self"; method:"GET",
<  <https://api.fakecompany.com/v1/addresses/42>; rel="addresses"; method:"GET",
<  <https://api.fakecompany.com/v1/orders/1234>; rel="orders"; method:"GET"
```

"Non-Resources" scenarios

D'après la théorie RESTful, toute requête doit être vue et manipulée comme une ressource. Or en pratique, ce n'est pas toujours possible, surtout lorsqu'on parle d'opération comme la

traduction, le calcul, la conversion, ou des services métiers parfois complexes inhérents à un SI.

Dans ces cas là, votre opération doit être représentée par un verbe et non un nom. Par exemple:

```
POST /calculator/sum
[1,2,3,5,8,13,21]
< 200 OK
< {"result" : "53"}
```

Ou encore :

```
POST /convert?from=EURato=USD&amount=42
< 200 OK
< {"result" : "54"}
```

On en vient donc à devoir utiliser des actions et non des ressources. Pour cela, on utilisera le verbe POST du protocole HTTP.

```
CURL –X POST \
-H "Content-Type: application/json" \
https://api.fakecompany.com/v1/users/42/carts/7/commit
< 200 OK
< { "id_cart": "7",<i> [...]</i> }
```

Pour correctement appréhender cette exception à la modélisation de votre API, le plus simple et de partir du principe que toute requête POST est une action, ayant un verbe par défaut lorsque celui ci n'est pas spécifié.

Dans le cas d'une collection de ressources par exemple, l'action par défaut est la création:

```
POST /users/create                    ==    POST /users
< 201 OK                                    < 201 OK
< { "id_user": 42 }                         < { "id_user": 42 }
```

Dans le cas d'un ressource "email", l'action par défaut sera son envoi au destinataire:

```
POST /emails/42/send                  ==    POST /emails/42
< 200 OK                                    < 200 OK
< { "id_email": 42, "state": "sent" }       < { "id_email": 42, "state": "sent" }
```

Il est cependant indispensable de garder à l'esprit que l'utilisation de verbe dans vos requêtes REST doit rester une exception. Et que dans la grande majorité des cas cette exception doit être évitée. Le fait qu'une ressource possède plusieurs actions, ou que de nombreuse ressources exposent au moins une action, est une alerte concernant le design de votre API.

Cela signifie généralement que vous avez adopté une approche RPC plutôt que REST, il sera donc nécessaire de prendre rapidement des actions pour reprendre le design de votre API en main.

Afin d'éviter de créer toute confusion pour les développeurs entre les ressources (sur lesquelles on peut faire du CRUD) et les opérations, il est fortement conseillé de distinguer ces deux parties dans votre documentation développeur.

➡ Chez les Géants du Web

API	"Non Resources" API
Google Translate API	GET https://www.googleapis.com/language/translate/v2?key=INSERT-YOUR-KEY&target=de&q=Hello%20world
Google Calendar API	POST https://www.googleapis.com/calendar/v3/calendars/calendarId/clear
Twitter Authentification	GET https://api.twitter.com/oauth/authenticate?oauth_token=Z6eEdO8MOmk394WozF5oKyuAv855l4Mlqo7hhlSLik

Erreurs

Structure d'erreur

Nous préconisons d'utiliser la structure *Json* suivante :

```
{
  "error": "description_courte",
  "error_description": "description longue, Human-readable",
  "error_uri": "URI vers une description détaillée de l'erreur sur le portail developper"
}
```

L'attribut error n'est pas forcement redondant avec le statut HTTP : il est possible d'avoir deux

statuts différents pour une même valeur sur la clé « error » et inversement.

- 400 & error=invalid_user
- 400 & error=invalid_cart

Cette représentation est issue de la spécification OAuth2. Sa généralisation à l'API permettra au client qui la consomme de ne pas avoir à gérer deux structures d'erreurs distinctes.

Note : il peut être pertinent de fournir dans certains cas une collection de cette structure, pour retourner plusieurs erreurs simultanées (utile dans le cas d'une validation formulaire *server-side* par exemple).

Status Codes

Nous préconisons fortement d'utiliser les codes de retour HTTP, de manière appropriée, sachant qu'il existe un code pour chaque cas d'utilisation courant. Ces codes sont connus de tous. Il n'est pas forcément nécessaire d'utiliser l'intégralité des codes, mais la douzaine de codes les plus utilisés est bien souvent suffisante.

SUCCESS

200 OK: Code de succès classique, fonctionnant dans les principaux cas. Spécialement uitilisé lors d'une première requête GET réussie sur une ressource.

HTTP Status	Description
201 Created	Indique qu'une ressource a été créé. C'est la réponse typique aux requête PUT et POST, en incluant une en-tête HTTP "Location" vers l'url de la ressource.
202 Accepted	Indique que la requête a été acceptée pour être traitée ultérieurement. C'est la réponse typique à un appel asynchrone (pour une meilleure UX ou de meilleures performances, ...).
204 No Content	Indique que la requête a été traitée avec succès, mais qu'il n'y a pas de réponse à retourner. Souvent retourné en réponse à un DELETE.
206 Partial Content	La réponse est incomplète. Typiquement retourné par les réponses paginées.

CLIENT ERROR

HTTP Status	Description
400 Bad Request	Généralement utilisé pour les erreurs d'appels, si aucun autre status ne correspond. On peut distinguer deux types d'erreurs. Request behaviour error, example GET /users?payed=1 < 400 Bad Request < {"error": "invalid_request", "error_description": "There is no 'payed' property on users."} Application condition error, example POST /users {"name":"John Doe"} < 400 Bad Request < {"error": "invalid_user", "error_description": "A user must have an email adress"}
401 Unauthorized	Je ne vous connais pas, dites moi qui vous êtes et je vérifierai vos habilitations. GET /users/42/orders < 401 Unauthorized < {"error": "no_credentials", "error_description": "This resource is under permission, you must be authenticated with the right rights to have access to it" }
403 Forbidden	Vous êtes correctement authentifié, mais vous n'êtes pas suffisamment habilité. GET /users/42/orders < 403 Forbidden < {"error": "not_allowed", "error_description": "You're not allowed to perform this request"}
404 Not Found	La ressource que vous demandez n'existe pas. GET /users/999999/ < 400 Not Found < {"error": "not_found", "error_description": "The user with the id '999999' doesn't exist" }

HTTP Status	Description
405 Method not allowed	Soit l'appel d'une méthode n'a pas de sens sur cette ressource, soit l'utilisateur n'est pas habilité à réaliser cette appel. POST /users/8000 < 405 Method Not Allowed < {"error":"method_does_not_make_sense", "error_description":"How would you even post a person?"}
406 Not Acceptable	Rien ne match au Header Accept-* de la requête. Par exemple, vous demandez une ressources XML or la ressources n'est disponnible qu'en Json. GET /usersAccept: text/xmlAccept-Language: fr-fr < 406 Not Acceptable < Content-Type: application/json < {"error": "not_acceptable", "available_languages":["us-en", "de", "kr-ko"]}

SERVER ERROR

HTTP Status	Description
500 Server error	L'appel de la ressource est valide, mais un problème d'exécution est rencontré. Le client ne peut pas réellement faire quoi que ce soit à ce propos. Nous proposons de retourner systématiquement un Status 500. GET /users < 500 Internal server error < Content-Type: application/json < {"error":"server_error", "error_description":"Oops! Something went wrong..."}

Sources

- Design Beautiful REST + JSON APIs
  - http://www.slideshare.net/stormpath/rest-jsonapis
- Web API Design: Crafting Interfaces that Developers Love
  - https://pages.apigee.com/web-api-design-website-h-ebook-registration.html

- HTTP API Design Guide
  - https://github.com/interagent/http-api-design

- RESTful Web APIs
  - http://shop.oreilly.com/product/0636920028468.do

- How are REST APIs versioned?
  - http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/

- REST World
  - http://nodejsparis.bitbucket.org/20140312/rest_world/#/

Cet article a été posté dans Archi & techno et taggué API, REST, WOA.

OCTO Technology
Part of Accenture Digital

PARIS | RABAT | LAUSANNE | SYDNEY

Siège: 34 avenue de l'Opéra, 75002 Paris, France | +33 (0)1 58 56 10 00