I - Présentation des WS REST

1. Deux grands types de WS (REST et SOAP)

2 grands types de services WEB: **SOAP/XML** et **REST/HTTP**

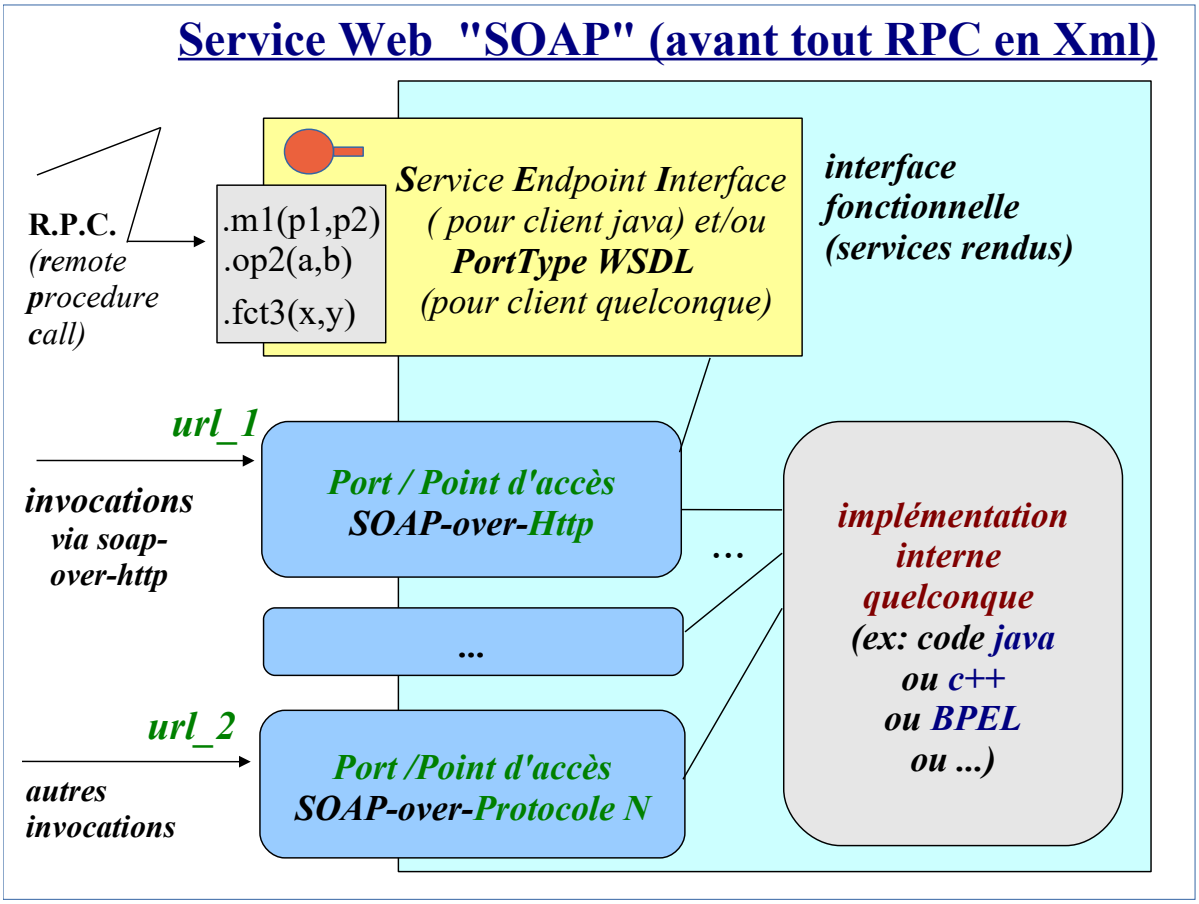
WS- (SOAP / XML)*

- **"Payload"** systématiquement en **XML** *(sauf pièces attachées / HTTP)*
- **Enveloppe SOAP** en XML *(header facultatif pour extensions)*
- **Protocole de transport au choix (HTTP, JMS, ...)**
- Sémantique quelconque (*appels méthodes*) , **description WSDL**
- **Plutôt** orienté Middleware SOA **(arrière plan)**

REST (HTTP)

- **"Payload" au choix (XML , HTML , JSON, ...)**
- Pas d'enveloppe imposée
- **Protocole de transport = toujours HTTP.**
- Sémantique **"CRUD" (*modes http* PUT,GET,POST,DELETE)**
- **Plutôt** orienté IHM Web/Web2 **(avant plan)**

1.1. Caractéristiques clefs des web-services SOAP / Xml



Points clefs des Web services "SOAP"

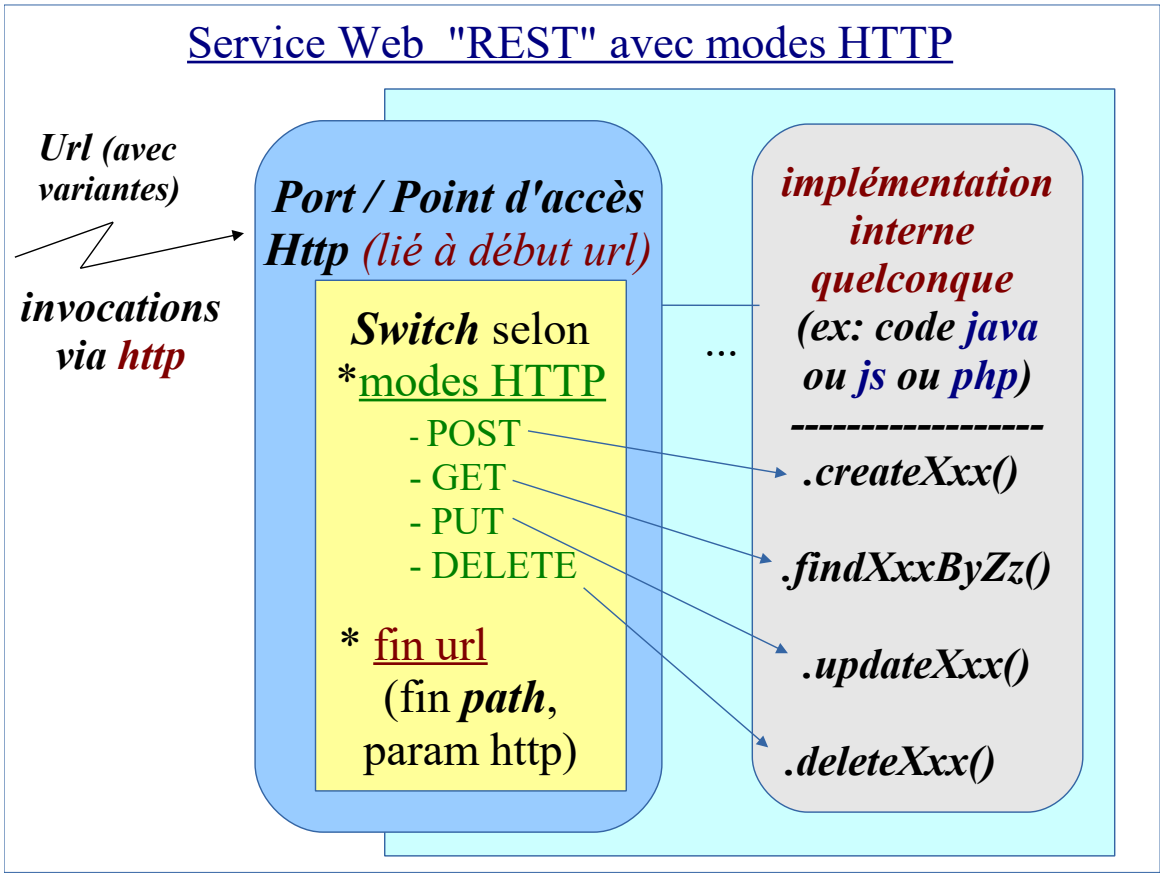
Le format "xml rigoureux" des requêtes/réponses (définis par ".xsd" , ".wsdl") permet de retraiter sans aucune ambiguïté les messages "soap" au niveau certains services intermédiaires (dans ESB ou ...). Certains automatismes génériques sont applicables .

Fortement typés (xsd:string , xsd:double) les web-services "soap" conviennent très bien à des appels et implémentations au sein de langages fortement typés (ex : "c++" , "c#" , "java" , "…").

A l'inverse , des langages faiblement typés tels que "php" ou "js" sont moins appropriés pour soap (appels cependant faisables)

La **relative complexité et la verbosité "xml" des messages "soap" engendrent des appels moins immédiats (*mode http "post" avec enveloppe à préparer"*) et des performances moyennes.**
Soap peut être utilisé en mode "envoi de document" mais c'est rare.
Les messages "soap" peuvent être véhiculés par "jms" mais c'est rare.

1.2. Caractéristiques clefs des web-services "REST" / "HTTP"



Points clefs des Web services "REST"

Retournant des données dans un format quelconque ("XML" , "JSON" et éventuellement "txt" ou "html") les web-services "REST" offrent des résultats qui nécessitent généralement peu de re-traitements pour être mis en forme au sein d'une IHM web.

Le format "au cas par cas" des données retournées par les services REST permet peu d'automatisme(s) sur les niveaux intermédiaires.

Souvent associés au format "JSON" les web-services "REST" conviennent parfaitement à des appels (ou implémentations) au sein du langage javascript .

La relative simplicité des URLs d'invocation des services "REST" permet des appels plus immédiats (*un simple href="..." suffit en mode GET pour les recherches de données*) .

La compacité/simplicité des messages "JSON" souvent associés à "REST" permet d'obtenir d'assez bonnes performances .

2. Web Services "R.E.S.T."

REST = style d'architecture (conventions)

REST est l'acronyme de **Re**presentational **S**tate **T**ransfert.
C'est un **style d'architecture** qui a été décrit par ***Roy Thomas Fielding*** dans sa thèse *«Architectural Styles and the Design of Network-based Software Architectures»*.

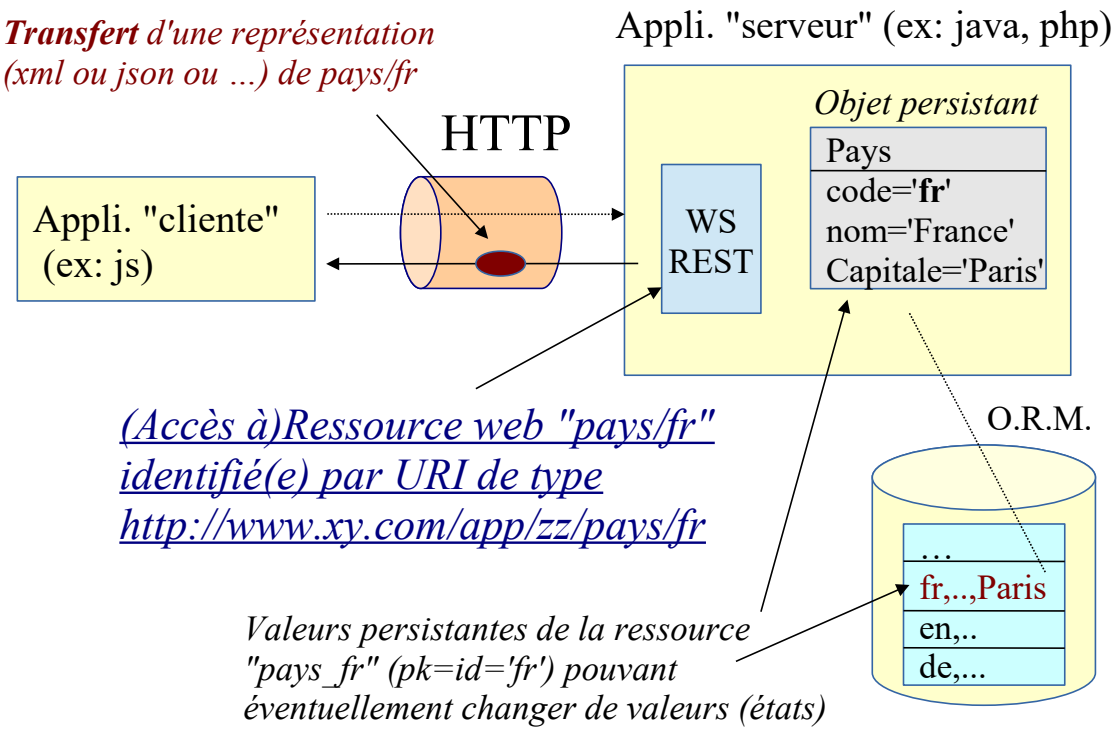
L'information de base, dans une architecture REST, est appelée **ressource**.
Toute information (à sémantique stable) qui peut être nommée est une ressource: un article , une photo, une personne , un service ou n'importe quel concept.

Une ressource est identifiée par un **identificateur de ressource**. Sur le web ces identificateurs sont les **URI** (Uniform Resource Identifier).

NB: dans la plupart des cas, une ressource REST correspond indirectement à un enregistrement en base (avec la *clef primaire* comme partie finale de l'uri "identifiant").

Les composants de l'architecture REST manipulent ces ressources en **transférant à travers le réseau** (via HTTP) des **représentations de ces ressources**.
Sur le web, on trouve aujourd'hui le plus souvent des représentations au format **HTML , XML ou JSON**.

REST : transferts de représentations de ressources



REST et principaux formats (xml,json)

Une invocation d'URL de service REST peut être accompagnée de données (en entrée ou en sortie) pouvant prendre des formats quelconques :
text/plain , text/html , application/xml , application/json , …

Dans le cas d'une lecture/recherche d'informations , le format du résultat retourné pourra (selon les cas) être :

- **imposé (en dur) par le code du service REST** .

- **au choix (xml , json) et précisé par une partie de l'url**

- **au choix (xml , json) et précisé par le champ "Accept :" de l'entête HTTP de la requête**. (exemple: Accept: application/json) .

Dans tous les cas, la réponse HTTP devra avoir son format précisé via le champ habituel ***Content-Type:*** *application/json* de l'entête.

Format JSON (JSON = *JavaScript Object Notation*)

Les 2 principales caractéristiques de JS0N sont :

- Le principe de clé / valeur (map)
- L'organisation des données sous forme de tableau

Les types de données valables sont :

- tableau
- objet
- chaîne de caractères
- valeur numérique (entier, double)
- booléen (true/false)
- null

```
[
  {
    "nom": "article a",
    "prix": 3.05,
    "disponible": false,
    "descriptif": "article1"
  },
  {
    "nom": "article b",
    "prix": 13.05,
    "disponible": true,
    "descriptif": null
  }
]
```

une liste d'articles

une personne

```
{
  "nom": "xxxx",
  "prenom": "yyyy",
  "age": 25
}
```

REST et méthodes HTTP (verbes)

Les **méthodes HTTP** sont utilisées pour indiquer la **sémantique des actions demandées** :

- **GET** : **lecture/recherche** d'information
- **POST** : **envoi** d'information
- **PUT** : **mise à jour** d'information
- **DELETE** : **suppression** d'information

Par exemple, pour récupérer la liste des adhérents d'un club, on peut effectuer une requête de type **GET** vers la ressource **http://monsite.com/adherents**

Pour obtenir que les adhérents ayant plus de 20 ans, la requête devient http://monsite.com/adherents?ageMinimum=20

Pour supprimer numéro 4, on peut employer une requête de type **DELETE** telle que **http://monsite.com/adherents/4**

Pour envoyer des informations, on utilise **POST** ou **PUT** en passant les informations dans le corps (invisible) du message HTTP avec comme URL celle de la ressource web que l'on veut créer ou mettre à jour.

Exemple concret de service REST : "Elevation API"

L'entreprise "***Google***" fourni gratuitement certains services WEB de type REST.
"***Elevation API***" est un service REST de Google qui renvoie l'altitude d'un point de la planète selon ses coordonnées (latitude ,longitude) .
La documentation complète se trouve au bout de l'URL suivante :

https://developers.google.com/maps/documentation/elevation/?hl=fr

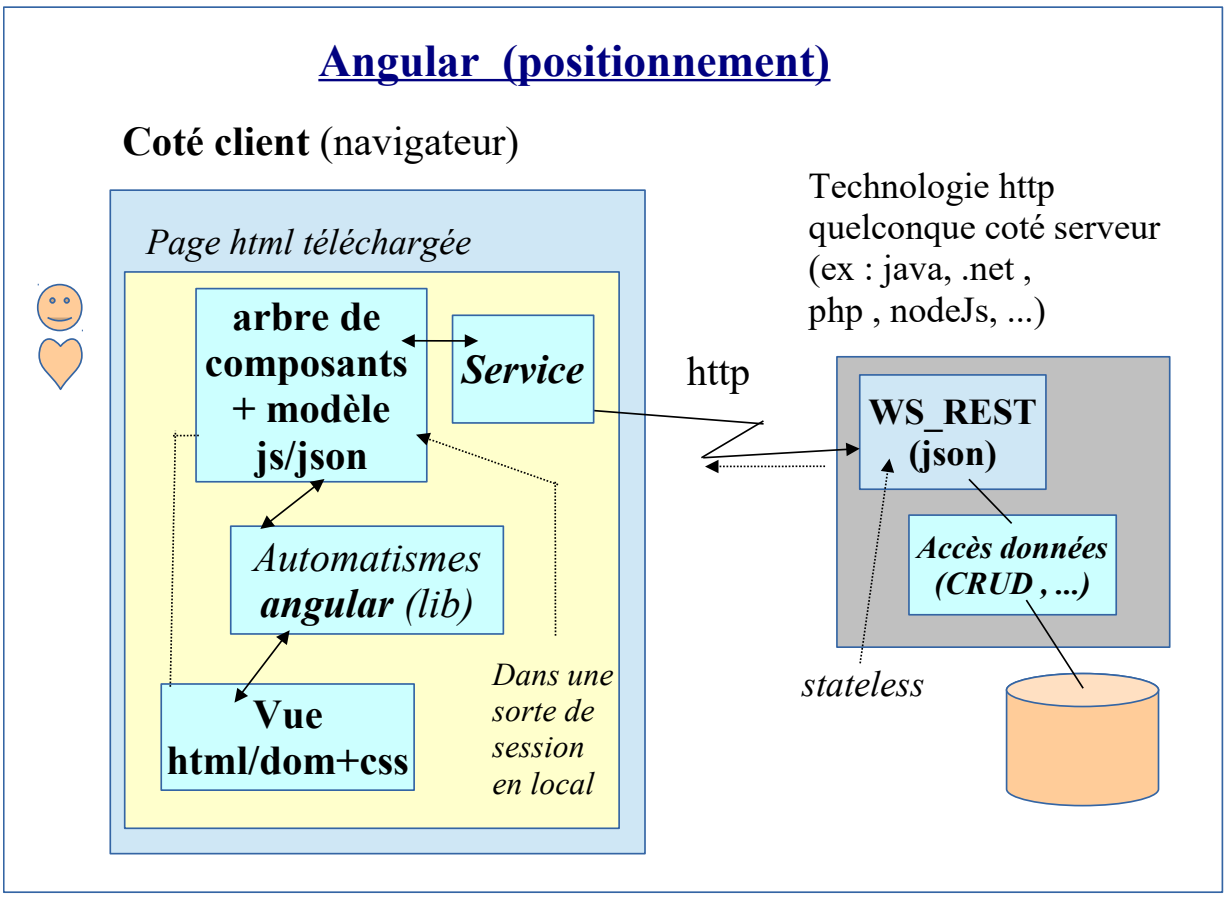
Sachant que les coordonnées du Mont blanc sont :
Lat/Lon : 45.8325 N / 6.86417 E *(GPS : 32T 334120 5077656)*
Les invocations suivantes (du service web rest "api/elevation")

http://maps.googleapis.com/maps/api/elevation/json?locations=45.8325,6.86417
http://maps.googleapis.com/maps/api/elevation/xml?locations=45.8325,6.86417

donne les résultats suivants "json" ou "xml":

```
{  "results" : [
     {
       "elevation" : 4766.466796875,
       "location" : {
         "lat" : 45.8325,
         "lng" : 6.86417
       },
       "resolution" : 152.7032318115234
     }
   ],   "status" : "OK"
}
```

```
?xml version="1.0" encoding="UTF-8"?>
<ElevationResponse>
 <status>OK</status>
 <result>
  <location>
   <lat>45.8325000</lat>
   <lng>6.8641700</lng>
  </location>
  <elevation>4766.4667969</elevation>
  <resolution>152.7032318</resolution>
 </result>
</ElevationResponse>
```



Conventions sur URL / Path des ressources REST

Type requêtes	HTTP *Method*	*URL ressource(s) distante(s)*	*Request body*	*Réponse JSON*
Recherche multiple	**GET**	**…/product** **…/product?crit1=v1&crit2=v2**	vide	Liste/tableau d'objets
Recherche par id	**GET**	**…/product/idRes** *(avec idRes=1,...)*	vide	Objet JSON
Ajout (seul)	**POST**	**…/product**	Objet JSON	Objet JSON avec id quelquefois calculé (incr)
Mise à jour (seule)	**PUT**	**…/product/idRes** **ou** **…/product**	Objet JSON avec **.id**	Objet JSON mis à jour
SaveOr Update	*POST*	*.../product*	*Objet JSON*	*Objet JSON ajouté (auto_incr id) ou modifié*
suppression	**DELETE**	**…/product/idRes**	vide	Statut et message
Autres	**...**	**…/product-action/opXy/....**	...	...

2.1. Statuts HTTP (code d'erreur ou ...)

Catégories de code/statut HTTP :

1xx	Information (rare)
2xx (ex : 200)	Succès
3xx	Redirection
4xx	Erreur du client
5xx (ex : 500)	Erreur du serveur

Principaux codes/statuts en cas de succès ou de redirection:

200 , OK	Requête traitée avec succès. La réponse selon méthode de requête utilisée
201 , Created	Requête traitée avec succès et création d'un document.
204 , No Content	Requête traitée avec succès mais pas d'information à renvoyer.
301 , *Moved Permanently*	Document déplacé de façon permanente
304 , *Not Modified*	Document non modifié depuis la dernière requête

Principaux codes d'erreurs :

400 , Bad Request	La syntaxe de la requête est erronée (ex : invalid argument)
401 , Unauthorized	Une authentification est nécessaire pour accéder à la ressource.
403, Forbidden	authentification effectuée mais manque de droits d'accès (selon rôles, ...)
404 , Not Found	Ressource non trouvée.
409 , Conflict	La requête ne peut être traitée en l'état actuel.
...	*liste complète sur https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP*
500 , Internal Server Error	Erreur interne (vague) du serveur (ex ; bug , exception , ...) .
501, Not Implemented	Fonctionnalité réclamée non supportée par le serveur
503 , Service Unavailable	Service temporairement indisponible ou en maintenance.

2.2. Variantes classiques

Réponses plus ou moins détaillées (simple "http status" ou bien "message json")

Lorsqu'un serveur répond à une requête en mode POST , il peut soit :

*- retourner le "https status" **201/CREATED** et une réponse JSON comportant toute l'entité sauvegardée coté serveur avec souvent l'id (clef primaire) automatiquement généré ou incrémenté*
*{ **"id" : "a345b6788c335d56"** , "name" : "toto" , ... }*

*- se contenter de renvoyer le "http status" **201/CREATED** avec aucun message de réponse mais avec le champ **Location: /type_entite/idxy** comportant au moins l'id de la resource enregistrée au sein de l'entête HTTP de la réponse .*
*L'application cliente pourra alors effectuer un second appel en mode GET avec une fin d'URL en **/type_entite/idxy** si elle souhaite récupérer tous les détails de l'entité sauvegardée .*

- combiner les 2 styles de réponses (champ Location ET réponse JSON)

Lorsqu'un serveur répond à une requête en mode DELETE , il peut soit :

- se contenter de renvoyer le "http status" **204/NO_CONTENT** et aucun message

- retourner le "https status" **200/0K** et une réponse JSON de type
{ "message" : "resource of id ... successfully deleted" }

Lorsqu'un serveur répond à une requête en mode PUT , il peut soit :

- se contenter de renvoyer le "http status" **204/NO_CONTENT** et aucun message

*- retourner le "https status" **200/0K** et une réponse JSON comportant toutes les valeurs de l'entité mise à jour du coté serveur , exemple:*
{ "id" : "a345b6788c335d56" , "name" : "titi" , ... }

*On peut éventuellement envisager que le serveur réponde **par défaut** aux modes PUT et DELETE par un simple **204/NO_CONTENT** et qu'il réponde par **200/OK + un message JSON** si le paramètre http optionnel **?v=true** ou **?verbose=true** est présent en fin de l'URL de la requête .*

Identifiant de la resource à modifier en mode PUT placé en fin d'URL ou bien dans le corps de la requête HTTP, ou bien les deux.

Lorsqu'un serveur reçoit une requête de mise à jour en mode PUT , l'id de l'entity peut soit être précisée en fin d'URL , soit être précisée dans les données json de la partie body et si l'information est renseignée des 2 façons elle ne doit pas être incohérente .

Le serveur peut éventuellement faire l'effort de récupérer l'id de l'une ou des deux façons envisageables et peut renvoyer ***400/BAD_REQUEST*** si l'id de l'entité à mettre à jour n'est pas renseigné ou bien incohérent.

2.3. Safe and idempotent REST API

Une Api "Rest" désigne un ensemble de Web-services liés à un certain domaine fonctionnel (ex : gestion des stocks ou facturation ou ...)

Un appel "HTTP" vers une api-rest est dit "***safe***" s'il n'engendre pas de modifications du coté des ressources du serveur (**"safe" = "readonly"**) .

En mathématique , une fonction est dite "idempotente" si plusieurs appels successifs avec les mêmes paramètres retournent toujours le même résultat.

Au niveau d'une **api-rest** , une **invocation HTTP** (***ex : GET , PUT ou DELETE***) est dite "**idempotente**" si **plusieurs appels successifs avec les mêmes paramètres engendrent un même "état résultat" au niveau du serveur** .

Mais la réponse HTTP peut cependant varier .

Exemple : premier appel à "delete xyz/567" --> return "200/OK" ou "204/NO_CONTENT"

et second appel à "delete xyz/567"--> return 404 / notFound

mais dans les 2 cas , la ressource de type "xyz" et d'id=567 est censée ne plus exister .

Le DELETE est donc généralement considéré comme idempotent .

	safe	idempotent
GET *(et HEAD,OPTIONS)*	y	y
PUT	n	y
DELETE	n	y
POST	n	n

Intérêt de l'impotence comportementale du coté serveur :

Une application cliente doit souvent passer par des intermédiaires pour véhiculer une requête HTTP jusqu'au serveur . Certains mécanismes intermédiaires considèrent "internet / http" comme pas fiable à 100 % et vont quelquefois effectuer plusieurs retransmissions d'une requête si la première tentative échoue . il vaut mieux donc que le serveur se comporte de manière idempotente dans un maximum de cas .

Bien que le vocabulaire "~~idempotence~~" ne soit pas du tout approprié , **il est tout de même conseillé de retourner des réponses HTTP dans un format assez homogène vers le client** pour que celui-ci soit simple à programmer (pas trop de if ... else ...)

Dans tous les cas , bien documenter "comportements & réponses" d'une apit rest .