REST (Representational State Transfer)

I. Mouakher

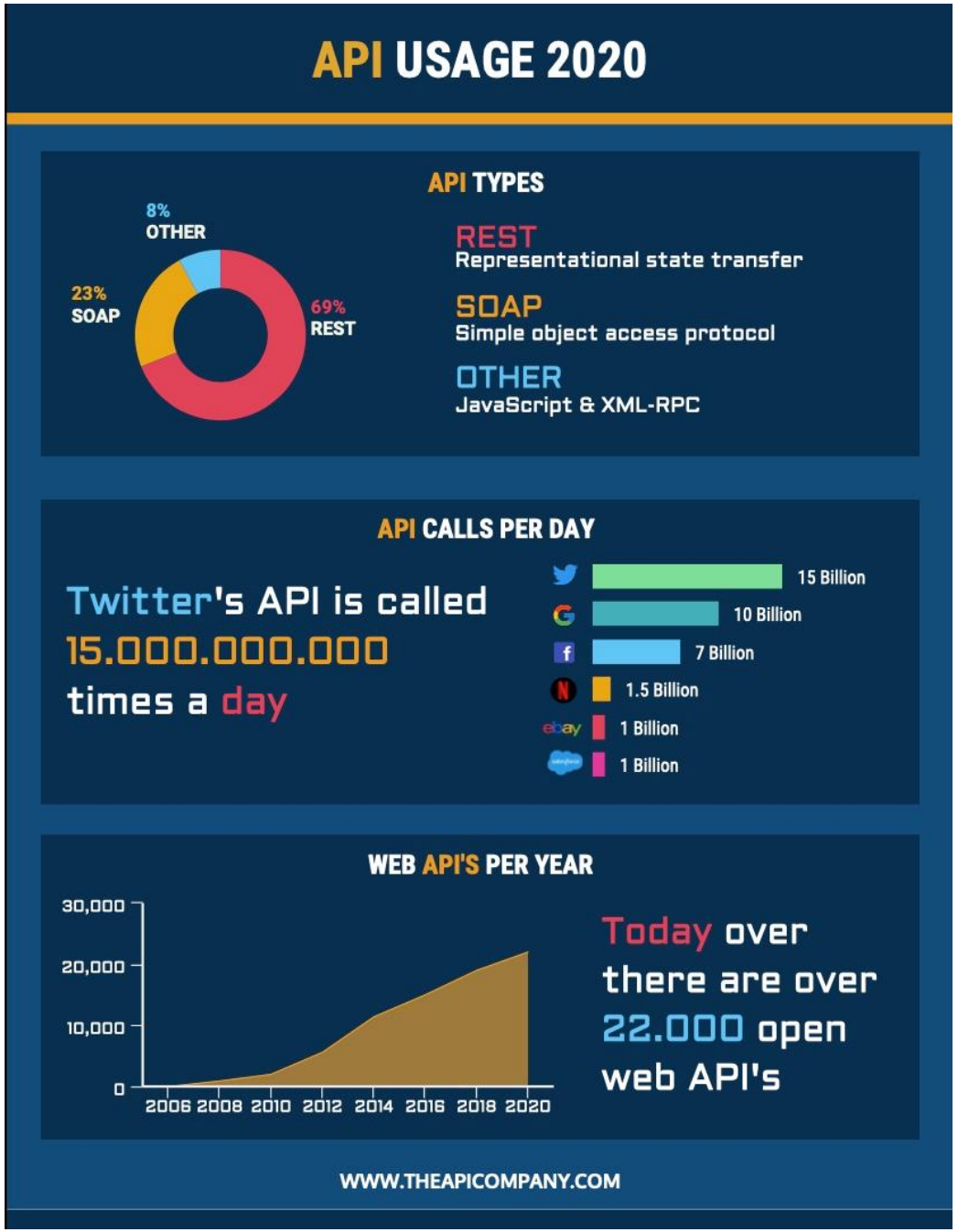
Types de services Web

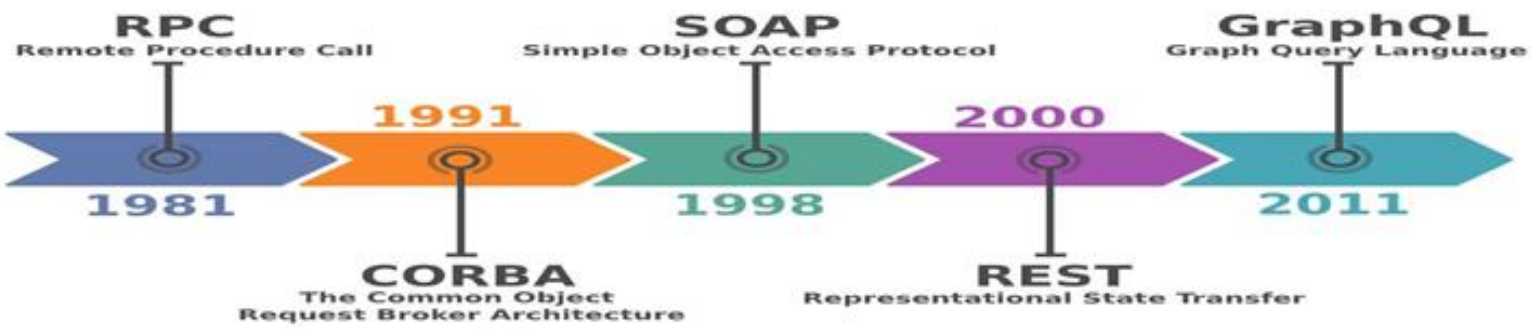
- **Etendus :** utilisent les standards **UDDI** (annuaire)/ **WSDL** (contrat) / **SOAP** (consommation)

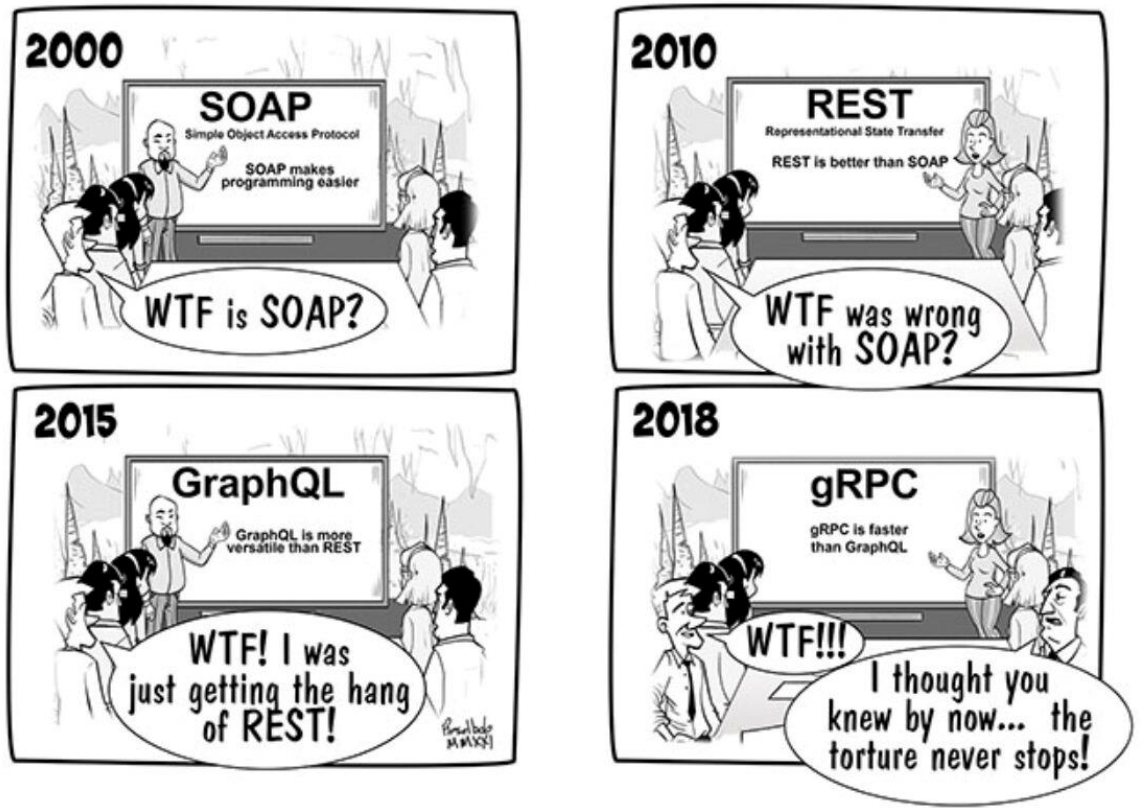
- **REST** (Representational State Transfer) utilisent :
  - directement **HTTP** au lieu d'une enveloppe SOAP
  - un **URI** pour nommer et identifier une ressource
  - les méthodes HTTP (**POST**, **GET**, **PUT** et **DELETE**) pour effectuer les opérations de base CRUD

« API » vs « service web »

- L’acronyme API (pour *Application Programming Interface*) signifie Interface de programmation d’applications.
- « API » et « Service web » désignent tous deux des moyens de communication
- Un service web facilite les interactions entre deux machines situées sur un réseau.
- Une API, elle, sert d’interface entre deux applications différentes, afin qu’elles puissent communiquer ensemble. C'est est une méthode par laquelle les fournisseurs tiers peuvent écrire des programmes pouvant facilement interagir avec d’autres programmes. Dans le cas d’applications web, l’API utilisée est basée sur le web.
  - Un service web est une API habillée d’HTTP


API USAGE 2020
API TYPES
8%
OTHER
23%
SOAP
69%
REST
REST
Representational state transfer
SOAP
Simple object access protocol
OTHER
JavaScript & XML-RPC
API CALLS PER DAY
Twitter's API is called
15.000.000.000
times a day
15 Billion
10 Billion
7 Billion
1.5 Billion
1 Billion
1 Billion
WEB API'S PER YEAR
30,000
20,000
10,000
0
2006 2008 2010 2012 2014 2016 2018 2020
Today over
there are over
22.000 open
web API's
WWW.THEAPICOMPANY.COM


RPC
Remote Procedure Call
1981
1991
CORBA
The Common Object
Request Broker Architecture
SOAP
Simple Object Access Protocol
1998
2000
REST
Representational State Transfer
GraphQL
Graph Query Language
2011


2000
SOAP
Simple Object Access Protocol
SOAP makes programming easier
WTF is SOAP?
2010
REST
Representational State Transfer
REST is better than SOAP
WTF was wrong with SOAP?
2015
GraphQL
GraphQL is more versatile than REST
WTF! I was just getting the hang of REST!
2018
gRPC
gRPC is faster than GraphQL
WTF!!!
I thought you knew by now... the torture never stops!

REST

- Rest (Representation State Transfert) ou Restful est un *style d'architecture* pour les systèmes hypermédia distribués
- Il s'agit d'un ensemble de conventions et de bonnes pratiques à respecter et non d'une technologie à part entière
- L'architecture Rest utilise les spéccifications originelles du protocole HTTP, plutôt que de réinventer une surcouche (comme le font SOAP ou XML-RPC par exemple)

RESTful

- Une application est dite RESTful lorsqu'elle respecte six recommandations architecturales :
  1. **Uniform interface.** Une interface uniforme entre les composants qui permet un transfert standardisé des informations au lieu d'un échange personnalisé en fonction des besoins d'une application. Roy Fielding, le créateur de REST, décrit ceci comme « la fonction centrale qui distingue le style architectural REST des autres styles basés sur le réseau ».
  2. **Client-server architecture.** Une architecture client-serveur composée de clients, de serveurs et de ressources
  3. **Statelessness.** Des communications client-serveur sans état, ce qui signifie que le contenu du client n'est jamais stocké sur le serveur entre les requêtes ; les informations sur l'état de la session sont stockées sur le client
  4. **Cacheability.** Des données qui peuvent être mises en mémoire cache pour éviter certaines interactions entre le client et le serveur
  5. **Layered system**. Un système à couches où des couches hiérarchiques peuvent assurer la médiation dans les interactions entre le client et le serveur
  6. **Code on demand (optional).** Du code à la demande qui permet au serveur d'étendre la fonctionnalité d'un client en transférant le code exécutable (recommandation facultative, car elle réduit la visibilité). (Most of the time, you will be sending the static representations of resources in the form of XML or JSON. But when you need to, you are free to return executable code to support a part of your application, e.g., clients may call your API to get a UI widget rendering code. It is permitted.)

Functionality as a set of resources

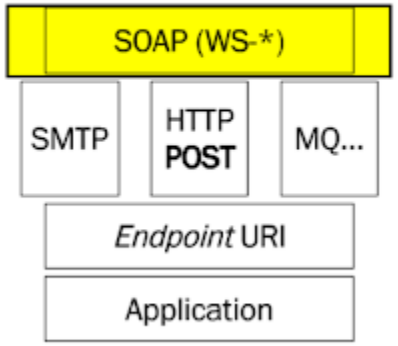
- A RESTful application is an application that exposes its state and functionality as a set of resources that the clients can manipulate and conforms to a certain set of principles:
  - All resources are uniquely addressable, usually through URIs; other addressing can also be used, though.
  - All resources can be manipulated through a constrained set of well-known actions, usually CRUD (create, read, update, delete), represented most often through the HTTP's POST, GET, PUT and DELETE; it can be a different set or a subset though - for example, some implementations limit that set to read and modify only (GET and PUT) for example
  - The data for all resources is transferred through any of a constrained number of well-known representations, usually HTML, XML or JSON;
  - The communication between the client and the application is performed over a stateless protocol that allows for multiple layered intermediaries that can reroute and cache the requests and response packets transparently for the client and the application.

SOAP vs REST

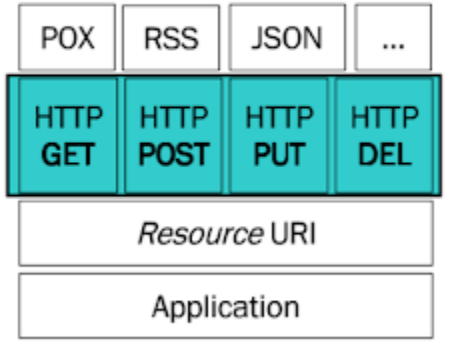
Protocole

Style d'architecture

SOAP



REST



SOAP vs REST

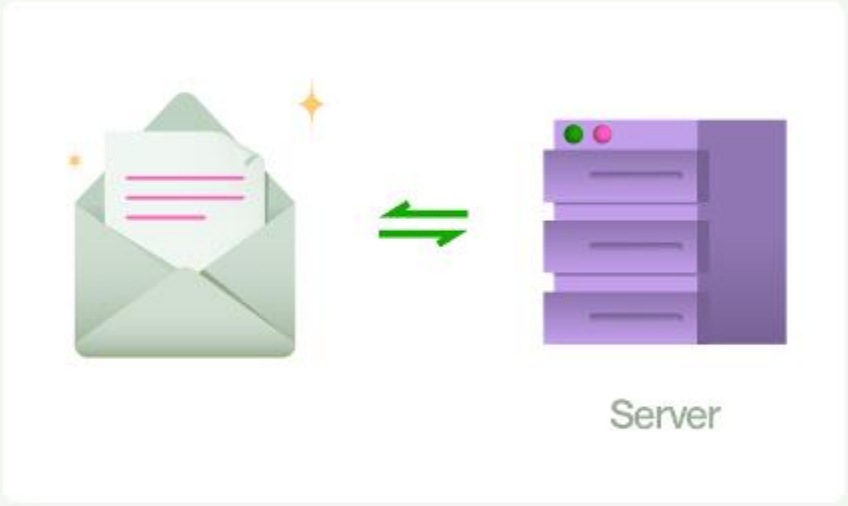
SOAP	REST
est un protocole officiel géré par le W3C (World Wide Web Consortium).	Est un ensemble de principes architecturaux
Comme il s'agit d'un protocole, il impose des règles intégrées qui augmentent la complexité et les coûts, ce qui peut ralentir le chargement des pages. SOAP services sont plus complexes à implementer et à consommer.	Les API RESTful sont plus flexibles et plus faciles à mettre en place.
Les services SOAP ont une structure et une interface bien définies (WSDL) et ont un ensemble de normes bien définies (WS-Security, WS-AtomicTransaction, and WS-ReliableMessaging). Ces standards assurent la conformité et sont ainsi privilégiés pour certains scénarios d'entreprise.	Les normes de documentation avec REST évoluent
les messages SOAP doivent être renvoyés sous la forme d'un document XML, un langage balisé lisible aussi bien par les humains que par les machines.	peuvent renvoyer des messages dans différents formats : HTML, XML, texte brut et JSON. Le format JSON (JavaScript Object Notation) est le plus utilisé pour les messages, car, en plus d'être léger, il est lisible par tous les langages de programmation (en dépit de son nom) ainsi

SOAP vs REST

Lorsqu'une requête de données est envoyée à une API SOAP, elle peut être gérée par n'importe quel protocole de couches de l'application : HTTP (pour les navigateurs web), SMTP (pour les e-mails), TCP et autres.	L'envoi d'une requête de données à une API REST se fait généralement par le protocole HTTP
SOAP-based reads cannot be cached.	REST has better performance and scalability. REST reads can be cached.
De nombreux systèmes d'anciennes générations reposent encore sur le protocole SOAP.	REST est arrivé plus tardivement et est souvent considéré comme une solution plus rapide pour des scénarios basés sur le web.
Les services web SOAP intègrent des spécifications de sécurité et de conformité des transactions qui répondent aux besoins de nombreuses entreprises, mais qui les rendent également plus lourds.	Les API REST sont plus légères et donc plus adaptées aux concepts récents tels que l'Internet des objets (IoT), le développement d'applications mobiles et l'informatique sans serveur.

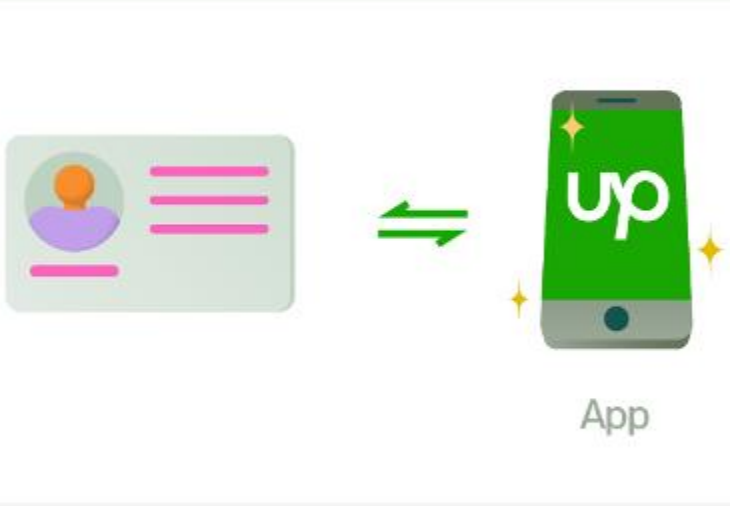
SOAP vs. REST APIs

upwork



SOAP is like using an envelope

Extra overhead, more bandwidth required, more work on both ends (sealing and opening).



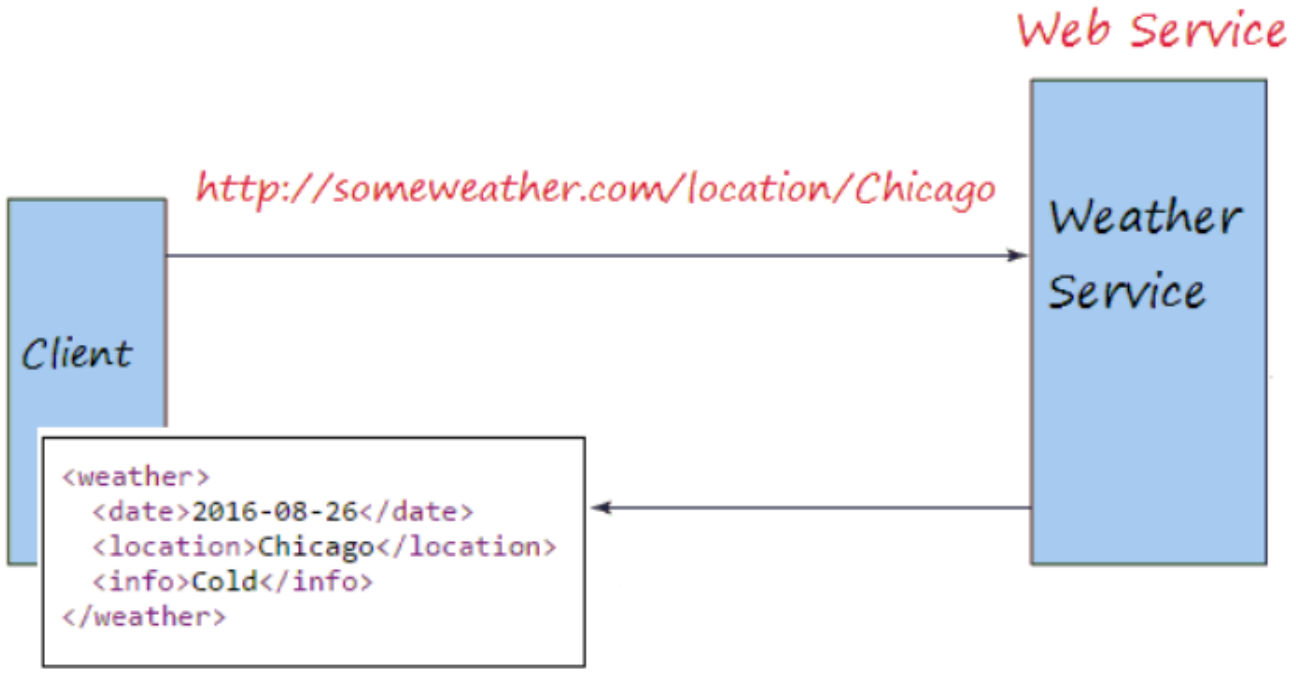
REST is like a postcard

Lighterweight, can be cached, easier to update.

Architectural constraints

Client-server architecture

- Généralement, le principe de la structure client-serveur est appliqué lorsque le serveur met un service à disposition qui peut être demandée par un client.



Statelessness (sans état)

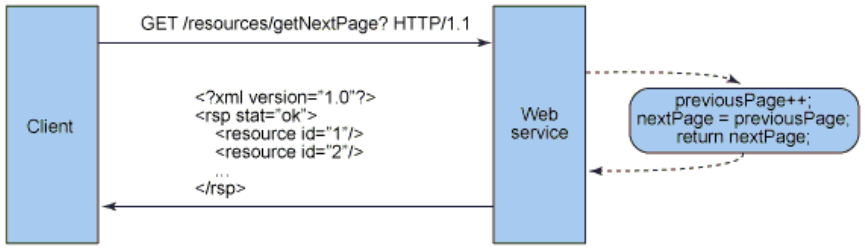
- La contrainte " *stateless*" signifie que tous les messages doivent inclure tout l'état de l'application.
- L'objectif du « *statelessness* »
  - Empêche les pannes partielles
  - Permet l'indépendance du substrate : L'équilibrage de charge ou Interruptions de service
- Les composants côté serveur sans état, par contre, sont moins compliqués à concevoir, à écrire et à distribuer sur des serveurs à répartition de charge équilibrée. Un service sans état non seulement fonctionne mieux, mais il transfère la plus grande partie de la responsabilité du maintien de l'état à l'application client. Dans un service Web RESTful, le serveur est responsable de générer des réponses et de fournir une interface qui permet au client de maintenir seul l'état de l'application.

Stateless: Exemple

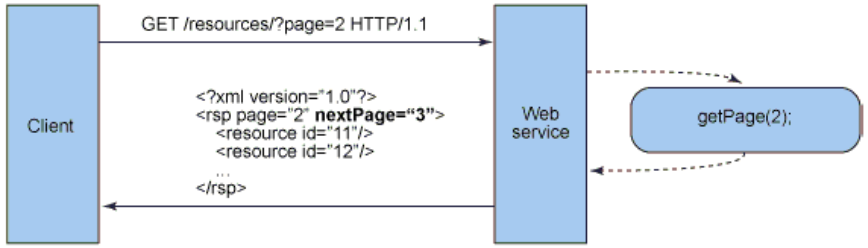
- Une caractéristique de REST est apatride, ce qui signifie qu'elle ne stocke pas les informations du client. Par exemple, vous venez d'envoyer une demande pour voir la page2 d'un document et vous souhaitez maintenant voir la page suivante (Page 3). REST ne stockera pas l'information qui vous a servi la page 2 précédemment. Cela signifie que REST ne gère pas Session.
- L'image ci-dessous illustre une application de stockage d'état qui sait quel numéro de page les utilisateurs visualisent. Et les utilisateurs ont besoin uniquement de demander la *"Next page"* pour obtenir la page désirée.

Stateless : Exemple

Une application de stockage d'état qui sait quel numéro de page les utilisateurs visualisent, les utilisateurs ont besoin uniquement de demander la "Next page" pour obtenir la page désirée.



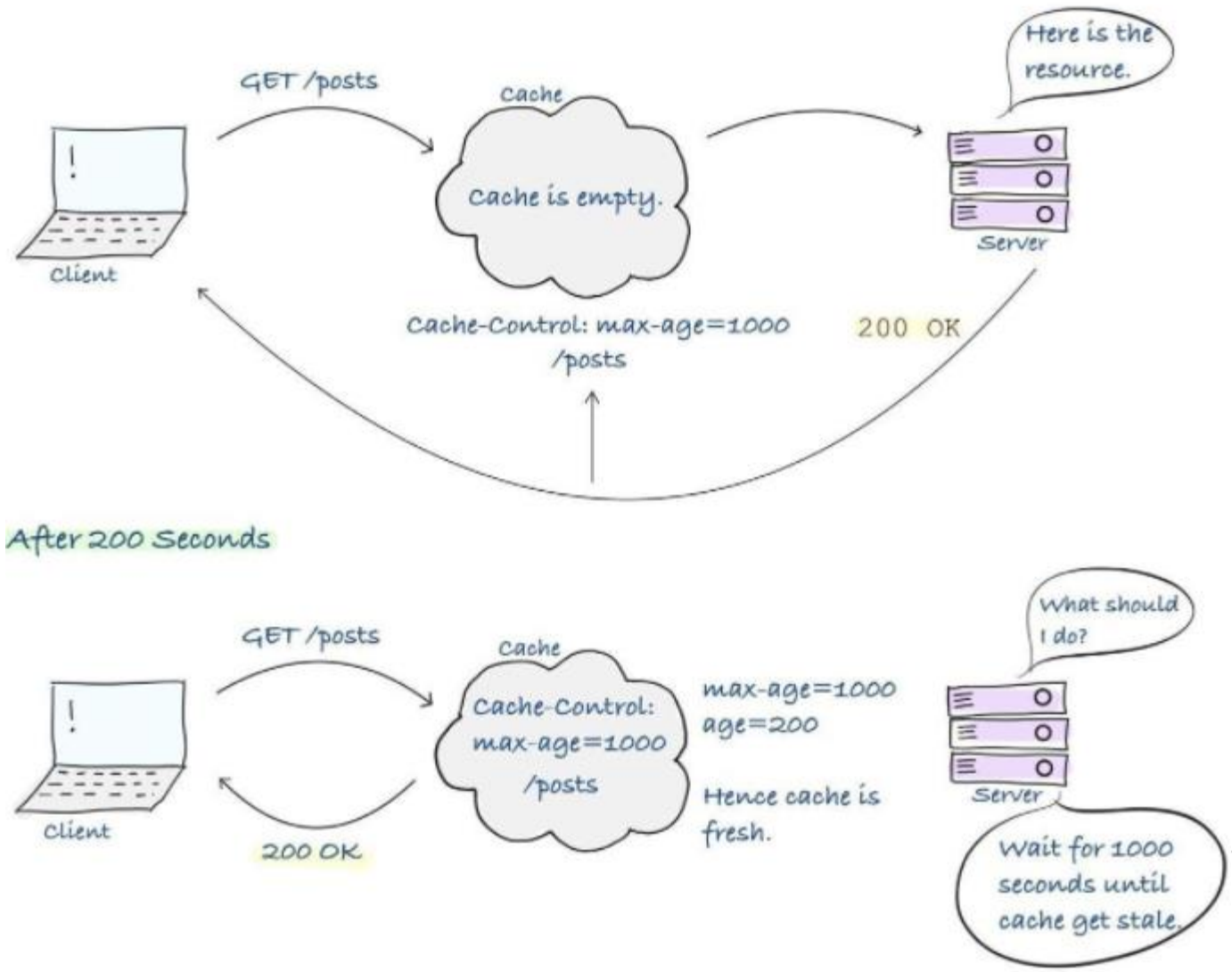
Pour les conceptions sans état, le client doit envoyer une exigence claire, y compris le numéro de page



Cacheable (ou *sauvegardable*, en français)

- La réponse doit contenir l'information sur la capacité ou non du client de mettre les données **en cache**, ou de les sauvegarder. Si les données **peuvent être mises en cache**, la réponse doit être accompagnée d'un numéro de version. Ainsi, si votre utilisateur formule deux fois la même requête (c'est-à-dire s'il veut revoir une page) et que les informations n'ont pas changé, alors votre serveur n'a pas besoin de rechercher les informations une deuxième fois. À la place, le client peut simplement mettre en cache les données la première fois, puis charger à nouveau les mêmes données la seconde fois.
- Une mise en cache efficace peut réduire le nombre de fois où un client et un serveur doivent interagir, ce qui peut aider à accélérer le temps de chargement pour l'utilisateur !
- Vous avez peut-être entendu le terme ***cache*** en référence à, par exemple, « Rafraîchissez le cache de votre navigateur ». Un cache est un moyen de **sauvegarder** des données pour pouvoir répondre plus facilement aux prochaines requêtes qui seront **identiques**. Quand vous allez sur de nombreux sites web depuis votre navigateur, celui-ci peut sauvegarder ces requêtes pour pouvoir compléter lui-même le site que vous voulez atteindre ou charger la page plus rapidement la prochaine fois que vous vous y rendez. Pratique !.

Caching in API calls



Code on demand (optional)

- Le code à la demande signifie que le serveur peut étendre sa fonctionnalité en envoyant le code au client pour téléchargement. C’est facultatif, car tous les clients ne seront pas capables de télécharger et d’exécuter le même code – donc ce n’est pas utilisé habituellement, mais au moins, vous savez que ca existe !

Functionality as a set of resources

Functionality as a set of resources

- les services web de type REST exposent entièrement ces fonctionnalités comme un ensemble de ressources identifiables par un URI et accessibles par la syntaxe et la sémantique du protocole HTTP. Les services web de type REST sont donc basés sur l'architecture du web et ses standards de base : HTTP et URI
- Il n'y a donc pas de différence fondamentale entre l'interaction d'un navigateur avec une ressource et celle d'un service web avec une ressource. La principale différence se situe au niveau du format de la représentation des données :HTML pour les navigateurs ou agents utilisateurs, XML ou JSON pour les services web ou agents ressources.
- On peut donc définir un service web comme l'implémentation logicielle d'une ressource, identifiée par un URI, et accessible en utilisant HTTP. Les agents s'occupent du contenu, de la représentation de leur état, pas du type de contenu. Il faut donc voir les services web comme le moyen de manipuler l'information, et non comme un simple fournisseur de services.

Functionality as a set of resources

Adressabilité

L'URI comme identifiant des resources
(Respecter un standard pour construire les URLs)

Interface homogène

Les méthodes HTTP comme identifiant des opérations
(POST, GET, PUT, DELETE)

Différentes représentations des ressources

Les réponses HTTP comme représentation des resources
Réponse HTTP en différents format (XML, JSON,..) en fonction de la demande de client

Hypermédia

Les liens comme relation entre resources
(href, rel)

Documentation

Adressabilité

- chaque ressource, par exemple, une commande, un produit ou un article doit pourvoir être identifié par un Unique Resource Identifier (URI).
- Une application se doit se construire ses URI (et donc ses URL) de manière précise, en tenant compte des contraintes REST.
- Les adresses REST service vent être intuitives au point d'être faciles à deviner. Pensez à une URI comme une sorte d'interface d'auto-documentation qui nécessite peu ou pas d'explications ou de références pour qu'un développeur comprenne ce qu'elle indique et puisse en tirer des ressources connexes. cette fin, la structure d'une URI doit être simple, prévisible et facile à comprendre.
- Les ressources sont regroupées dans un groupe que l'on appelle une collection. On s'y réfère avec la forme au pluriel du nom de la ressource.
- Il est nécessaire de prendre en compte la hiérarchie des ressources et la sémantique des URL pour les éditer
- Exemples :

Liste des livres

- GET http://mywebsite.com/books

Filtre et tri sur les livres

- GET http://mywebsite.com/books?fitlre=policier&tri=asc

Consulter un livre

- GET http://mywebsite.com/books/87

Tous les commentaires sur un livre

- GET http://mywebsite.com/books/87/comments

Ajoute un livre

- POST http://mywebsite.com/books

Mettre à ajour un livre

- PUT http://mywebsite.com/books/5

Supprimer un livre

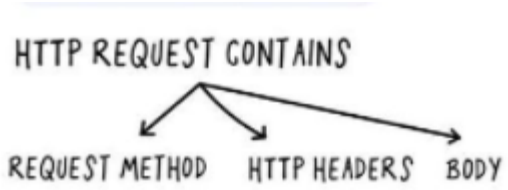
- DELETE http://mywebsite.com/books/5

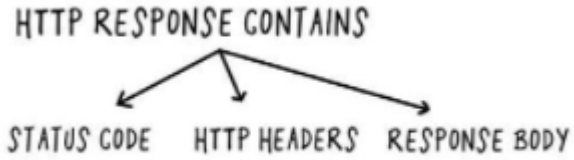
Interface homogène

- Interface homogène : chaque ressource doit pouvoir être facilement utilisée et de manière homogène, grâce à des méthodes standard. Par exemple avec les méthodes « GET », « POST » ou « PUT » en HTTP
- Utiliser les verbes HTTP existants plutôt que d'inclure l'opération dans l'URI de la ressource
- REST donne une règle obligeant les programmeurs à spécifier leur but via la méthode HTTP. Les finalités comprennent normalement l'obtention, l'insertion, la mise à jour ou l'effacement des données. Dans le cas où vous souhaitez réaliser l'un des objectifs, vous devez prendre note des règles ci-dessous :
  - Pour créer une ressource sur le serveur, vous devez utiliser la méthode POST.
  - Pour accéder à une ressource, utilisez GET.
  - Pour modifier l'état d'une ressource ou pour la mettre à jour, utilisez PUT.
  - Pour annuler ou supprimer une ressource, utilisez DELETE.

Message HTTP

- Les figures ci-dessous montrent les formats des messages request et response
- L'en-tête (Header) et le corps (Body) de la réponse contiennent des informations similaires à request, sauf que les informations sont différentes car il s'agit de la réponse du serveur.
- Le client peut faire un request en utilisant les méthodes http Le serveur renvoie une réponse avec un status code

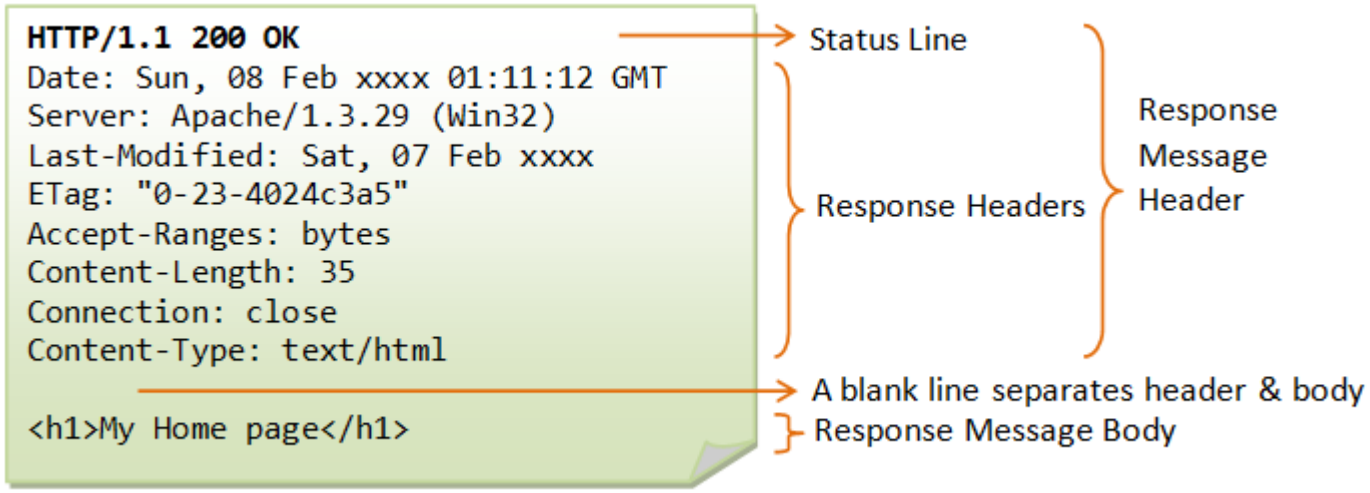




<VERB>: one of the HTTP methods	<URI> the resource URI to perform the operation	<HTTP Version>
<Request Header> contains information about the message such as format of the message body, format supported by client etc.		
<Request Body> holds the actual message content		

<HTTP Version>	<Response Code>
<Response Header>	
<Response Body>	

Message HTTP



Méthodes HTTP et REST

Méthode	Rôle	Code retour HTTP
GET URL	Récupération Element	200
GET URL	Récupération Collection	201
POST URL	Envoi d'Elements	201
DELETE URL	Effacer Element(s)	200
PUT URL	Modifier un Element	200
PATCH URL	Modif. partielle d'Elt.	200

Méthodes HTTP et REST (erreurs)

Code Erreur	Description	Signification
400	Bad Request	requête mal formée
404	Not Found	Resource demandée inexistante
401	Unauthorized	Authentification necessaire pour accéder à la resource.
405	Method Not Allowed	Méthode interdite pour cette resource.
409	Conflict	Par exemple, un PUT qui crée une ressource 2 fois
500	Internal Server Error	Autres erreurs du serveur.

Différentes représentations des ressources

- **C**haque ressource peut afficher différentes représentations. En fonction des exigences du client, différents langages ou formats comme HTML, JSON ou XML devront être utilisés par exemple.
- C’est le client de définir quel format de réponse il souhaite recevoir via l’entête http Accept

Format JSON vs XML

- **JSON** (JavaScript Object Notation – Notation Objet issue de JavaScript) est un format léger d'échange de données.
- Il est facile à lire ou à écrire pour des humains. Il est aisément analysable ou générable par des machines.
- JSON est un format texte complètement indépendant de tout langage, mais les conventions qu'il utilise seront familières à tout programmeur habitué aux langages descendant du C, comme par exemple : C lui-même, C++, C#, Java, JavaScript, Perl, Python et bien d'autres.
- Ces propriétés font de JSON un langage d'échange de données idéal.

```
// x est un objet java script
var x={a:4,b:9};
// T1 est un tableau java script
var T1=[8,6,8];
// T2 est un tableau d'objets
var T2=[{a:2,b:6},{a:1,b:8},{a:2,b:4}];
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
  <comptes>
      <compte type="courant">
          <code>1</code>
          <solde>4300</solde>
          <dateCreation>2012-11-11</dateCreation>
      </compte>
      <compte type="epargne">
          <code>2</code>
          <solde>96000</solde>
          <dateCreation>2012-12-11</dateCreation>
      </compte>
  </comptes>
```

JSON

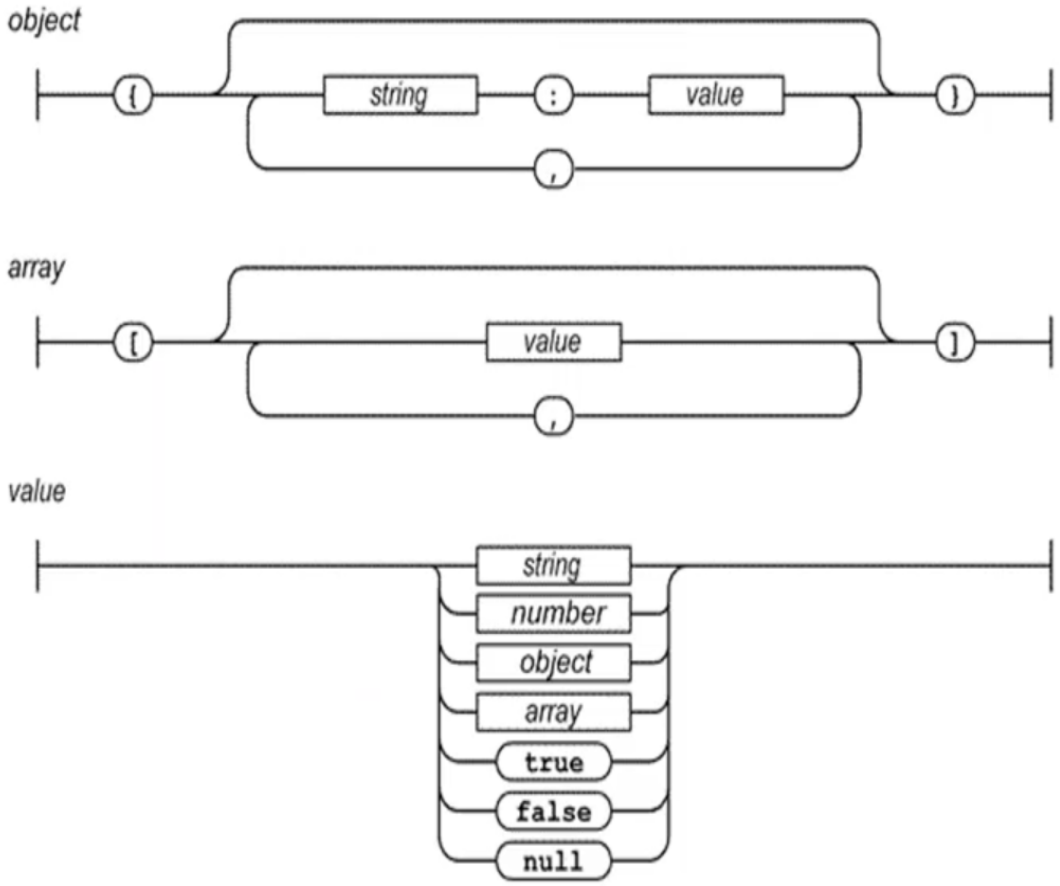
```
[
  {"type":"courant" , "code":1, "solde":4300,"dateCreation":1596711188451},
  {"type":"epargne" , "code":2, "solde":9600,"dateCreation":1596711179308}
]
```

Format JSON vs XML

- Il y a les schémas xsd il y a la garantit tous les données respecte un format semi structuré
- XML est un langage verbeux (nécessite beaucoup de text pèse sur la bande de communication)
- XML a besoin plus de resources pour son chargement en mémoir (parser)
- En Json l'application doit se charger de vérifier les formats
- Entre entreprise ont besoin d'avoir la garantit que les données respectent un contrat donc il préfère XML

Les stucture de données JSON

- En JSON, les structures de données prennent les formes suivantes :
- Un ***objet***, qui est un ensemble de couples nom/valeur non ordonnés. Un objet commence par { (accolade gauche) et se termine par } (accolade droite). Chaque nom est suivi de : (deux-points) et les couples nom/valeur sont séparés par , (virgule).
- Un ***tableau*** est une collection de valeurs ordonnées. Un tableau commence par [(crochet gauche) et se termine par] (crochet droit). Les valeurs sont séparées par , (virgule).
- Une ***valeur*** peut être soit une *chaîne de caractères* entre guillemets, soit un *nombre*, soit true ou false ou null, soit un *objet* soit un *tableau*. Ces structures peuvent être imbriquées.
- Une ***chaîne de caractères*** est une suite de zéro ou plus caractères Unicode, entre guillemets, et utilisant les échappements avec antislash. Un caractère est représenté par une chaîne d'un seul caractère.



Hypermédia (HATEOAS)

- **Hypermédia :** la mise à disposition des ressources a lieu via Hypermedia, par exemple sous forme d'attributs « href » et « src » dans des documents HTML ou pour l'interface définie des éléments JSON et XML. Ainsi, le client d'une API REST navigue uniquement avec des URLs mises à disposition par le serveur, « en fonction du principe Hypermedia as the Engine of Application State » (HATEOAS).
- Exemple quand on consulte une ressource qui représente un produit d'une catégorie, on trouve dans la description du produit un lien significatif qui permet de consulter la catégorie de ce produit.
-

Should you HATEOAS?

- There are a lot of mixed opinions as to whether the API consumer should create links or whether links should be provided to the API. RESTful design principles specify HATEOAS which roughly states that interaction with an endpoint should be defined within metadata that comes with the output representation and not based on out-of-band information.
- Although the web generally works on HATEOAS type principles (where we go to a website's front page and follow links based on what we see on the page), I don't think we're ready for HATEOAS on APIs just yet. When browsing a website, decisions on what links will be clicked are made at run time. However, with an API, decisions as to what requests will be sent are made when the API integration code is written, not at run time. Could the decisions be deferred to run time? Sure, however, there isn't much to gain going down that route as code would still not be able to handle significant API changes without breaking. That said, I think HATEOAS is promising but not ready for prime time just yet. Some more effort has to be put in to define standards and tooling around these principles for its potential to be fully realized.
- For now, it's best to assume the user has access to the documentation & include resource identifiers in the output representation which the API consumer will use when crafting links. There are a couple of advantages of sticking to identifiers - data flowing over the network is minimized and the data stored by API consumers is also minimized (as they are storing small identifiers as opposed to URLs that contain identifiers).
- Also, given this post advocates version numbers in the URL, it makes more sense in the long term for the API consumer to store resource identifiers as opposed to URLs. After all, the identifier is stable across versions but the URL representing it is not!
- (https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api)

Documentation

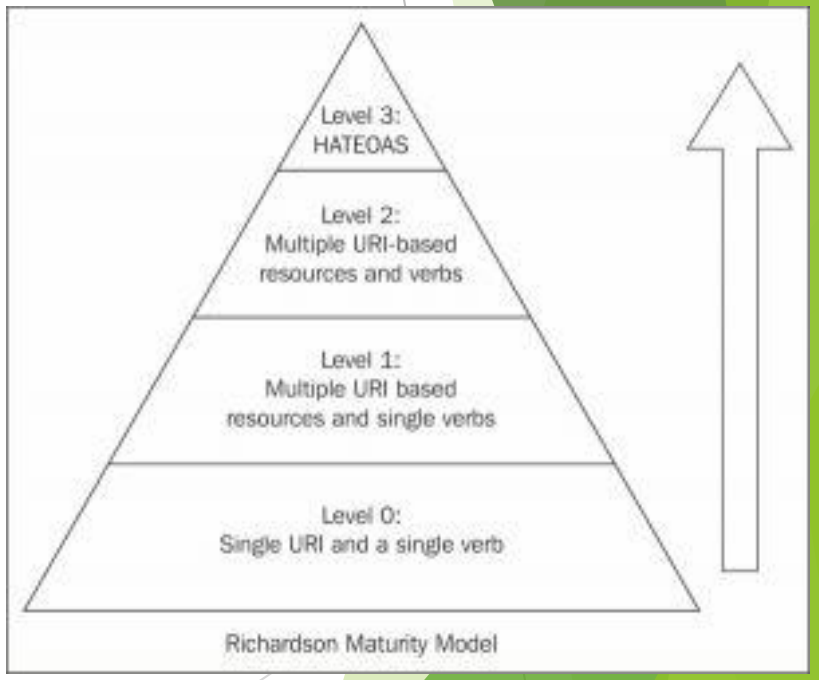
- An API is only as good as its documentation. The docs should be easy to find and publically accessible. Most developers will check out the docs before attempting any integration effort. When the docs are hidden inside a PDF file or require signing in, they're not only difficult to find but also not easy to search.
- The docs should show examples of complete request/response cycles. Preferably, the requests should be pastable examples - either links that can be pasted into a browser or curl examples that can be pasted into a terminal. GitHub and Stripe do a great job with this.
- Once you release a public API, you've committed to not breaking things without notice. The documentation must include any deprecation schedules and details surrounding externally visible API updates. Updates should be delivered via a blog (i.e. a changelog) or a mailing list (preferably both!).

Richardson Maturity Model

Richardson-maturity-model

- Il existe 4 niveaux d'API définis dans le modèle de maturité
  1. niveau 0: tout système ayant un seul point final pour tous ses apis (SOAP ou RPC font partie de cette catégorie).
  2. niveau 1: un système décrit par ResourceUri. C'est un système qui définit plusieurs URI basés sur des entités (au lieu d'avoir un seul sharepoint terminaison comme un système de niveau 0). Ces URI peuvent utiliser différentes actions http (POST, GET, PUT, etc.) pour mettre en œuvre différentes actions sur cette ressource.
  3. niveau 2: avec utilisation conforme des méthodes / verbes HTTP standard et réponses à plusieurs codes de statut
  4. niveau 3: plus HATEOAS (hypermedia inclus dans la réponse qui décrit les appels supplémentaires que vous pouvez faire)
- Alors que les niveaux 1, 2 et 3 peuvent être considérés comme des systèmes REST, seuls les niveaux 2 et 3 sont considérés comme RESTful.
- Donc, essentiellement tous les apis RESTful sont des apis REST, mais tous les apis REST ne sont pas RESTful

https://restfulapi.net/richardson-maturity-model/



REST API Features

Here is a list of features that makes it the most efficient way of data and application integration.

Scalability

REST API offers great scalability. As clients and servers are separated, it allows a product to be scaled by a team of developers without much trouble.

Plus, it is easier to integrate REST with present sites without refactoring website infrastructure. This allows developers to work faster instead of spending time reworking a website from scratch. As an alternative, they can merely add extra functionality. This makes it the most used method of integration.

Flexibility and Portability

Users can easily communicate even if the REST client-server is hosted on different servers, offering an important benefit from management's perspective.

Independence

Thanks to the parting between client and server, the REST protocol makes it easy for developments across the different areas of a project to occur autonomously. Moreover, the REST API is adjustable to the operational syntax and platform, offering the prospect to test numerous environments during development.

Références

- https://fr.wikipedia.org/wiki/Representational_state_transfer
- https://www.redhat.com/fr/topics/integration/whats-the-difference-between-soap-rest
- Thèse Roy Fielding:

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Acronymes

- **URI** (Universal Resource Identifiers): address used to identify a resource.
- **URL** (Universal Resource Locators): used to identify the resource and tell the protocol how locate and access the resource.