web API, API *RESTful* (21h)

Auteur: Paul Schuhmacher

Date de publication: 25/04/23

Date de dernière mise à jour : 06/07/23

Objectifs

- Comment *designer* une web API ?
- Quelle architecture utiliser ?
- Comment s'y prendre ?
- web API *RESTful*: principes, guides, bonnes pratiques
- Validation et échappement
- Sécuriser les accès aux ressources avec JWT Token
- Côté serveur: CORS
- Côté client: Single Origin Policy
- Comment documenter sa web API RESTFul ?

Définition d'une API web

Une API: *Application Programming Interface*. Une interface *programmable* d'une application

Une API web: une API exposée (ou partie exposée d'une application web) sur le web et basée sur le protocole HTTP.

> Comment *designer* une web API ? Quelle architecture utiliser ?

Retour aux fondamentaux

Surfing the web, le web est *simple*

Le web est un système *populaire*, car il permet à des personnes sans connaissances techniques de réaliser des choses utiles. Sur cette plateforme, un agent humain utilise les mêmes moyens qu'un agent programmable pour réaliser des actions.

> Le web *humain* est un sous-ensemble du web programmable.

Les technologies du web

- **Protocole HTTP**: protocole orienté *documents*
- **Adresses web (URL)**: chemin d'accès à une *ressource* distante. Une URL pointe sur une ressource.
- **Documents**: **HTML**, puis XML, JSON, etc.

Rappels sur le protocole HTTP

HTTP est strict en ce qui concerne le formatage des enveloppes, mais ne *se préoccupe pas de leur contenu*

Des documents dans des *enveloppes*

- Méthode (ou verbe) HTTP
- Chemin (URL/URI)
- En-têtes
- Corps (contenu de l'enveloppe)

Protocole HTTP: *Enveloppes* HTTP

Exemple d'une requête HTTP

```
GET /index.html HTTP/1.1
Host: www.oreilly.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.12)...
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,...
Accept-Language: us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-15,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Protocole HTTP: *Enveloppes* HTTP

Exemple d'une réponse HTTP

```
HTTP/1.1 200 OK
Date: Fri, 17 Nov 2006 15:36:32 GMT
Server: Apache
Last-Modified: Fri, 17 Nov 2006 09:05:32 GMT
Etag: "7359b7-a7fa-455d8264
Accept-Ranges: bytes
Content-Length: 43302
Content-Type: text/html
X-Cache: MISS from www.oreilly.com
Keep-Alive: timeout=15, max=1000
Connection: Keep-Alive

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
...
<title>oreilly.com -- Welcome to O'Reilly Media, Inc.</title>
...
```

Un peu d'histoire

- 1990: Tim Berners-Lee invente le web au CERN pour le partage de documents scientifiques. Il invente :
  - Les URI
  - HTTP
  - HTML
  - Le 1er serveur web
  - Le 1er navigateur web (WorldWideWeb puis Nexus)
- 6 août 1991: première page web publique
- 1996: le web compte 4 millions d'utilisateur·ices, x2 chaque mois, nombre de documents augmente de manière exponentielle. Le web va s'effondrer !

REST est issu de l'architecture même du web et de la specification HTTP

Un peu d'histoire

- 1993: Roy Fiedling et al. pose le problème de la montée en charge du web. Définit les 6 contraintes que le web doit respecter pour survivre (scaling)
  - Client serveur
  - Interface uniforme
  - Système en couches
  - Cache
  - Sans état
  - Code à la demande
- 1994: standardisation du protocole HTTP 1.1 avec Tim Berners Lee. Le web est *sauvé*

> Cet ensemble de contraintes sera rétrospectivement appelé REST (*Representational state transfer*) par Fiedling ou *Web's architectural style*

> Fiedling a sauvé le web du naufrage

Contraintes REST (*REpresentational State Transfer*)

REST (ou la *Web's architectural style*) :

REST n'est pas une architecture, mais un ensemble de contraintes définies notamment par Roy Fiedling lors de l'écriture de la spécification HTTP/1.1.

Les contraintes REST sont rangées en 6 catégories par Roy :

1. Client/Serveur
2. Interface Uniforme
    i. Adressabilité
    ii. HYATEOAS (connectivité)
    iii. Autodescriptif
    iv. Verbes (ou méthodes) HTTP
3. Système en couches
4. Cache
5. Sans état
6. Code à la demande

Une petite histoire: Alice à la découverte des contraintes du web

Une petite histoire pour introduire quelques concepts et contraintes (propriétés) du web :

- Concepts :
  - URL (URI)
  - Ressource
  - Représentation
- Contraintes :
  - **Adressabilité**
  - **Sans état** ou session courte

URI

```
http://www.youtypeitewepostit.com
```

URI : **U**niform **R**esource **I**dentifier

L'*URI* est le *nom* d'une ressource. Par définition, une URI pointe sur une ressource. On dit que la ressource est exposée. On parle de l'URI *de quelque chose*.

> L'**URL** (*Uniform Resource Locator*) est un URI qui en plus d'identifier une ressource (lui donner un nom unique) donne son emplacement sur le web (IP d'une machine + chemin). Par exemple, l'URL http://www.wikipedia.org/ est un URI qui identifie une ressource (page d'accueil Wikipédia) et donne également le moyen d'y accéder (machine enregistrée sous le nom de domaine www.wikipedia.org et le path '/'). Dans le cours on parlera d'URI et d'URL de manière interchangeable comme nous sommes dans un contexte web.

Ressource

> http://www.youtypeitewepostit.com semble être le nom de la ressource "Page d'accueil du site YouTypeItWePostIt". Tant qu'on a pas regardé, on ne peut pas savoir !

Une ressource peut être une œuvre d'art, un objet physique, un concept, un ensemble de références vers d'autres ressources, etc. Cela peut être *n'importe quoi*. C'est *conceptuel*. Chaque donnée intéressante que votre service manipule *devrait* être exposée comme une ressource.

Adressabilité ou l'identification unique des ressources

Un URI (ou URL pour le web) est un nom universel (et donc unique) donné à une ressource. **Chaque ressource doit avoir une URL**. Une ressource peut posséder une ou plusieurs URI. Une ressource doit posséder le moins de noms possible (dilution) et chaque nom devrait être significatif*

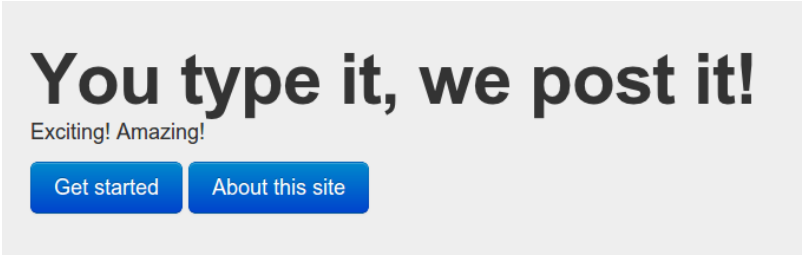
> *Cela dépend des approches. Tim Berners Lee plaide pour le fait qu'un URI ne devrait pas être nécessaire pour donner du contexte sur la ressource sous-jacente (appelé l'opacité des URI), d'autres non. Aujourd'hui on admet que donner des noms significatifs est une bonne chose, mais on ne devrait pas en avoir besoin pour explorer les ressources exposées par un serveur !

Alice consulte la ressource

```
GET / HTTP/1.1
Host: www.youtypeitwepostit.com
```

Le serveur répond à Alice

```
HTTP/1.1 200 OK
Content-type: text/html

<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <div>
          <h1>You type it, we post it!</h1>
          <p>Exciting! Amazing!</p>

          <p class="links">
            <a href="/messages">Get started</a>
            <a href="/about">About this site</a>
          </p>
        </div>
    </body>
</html>
```



Sans état (Stateless)

> Alice est "sur la page web du site". Mais le serveur a déjà oublié Alice

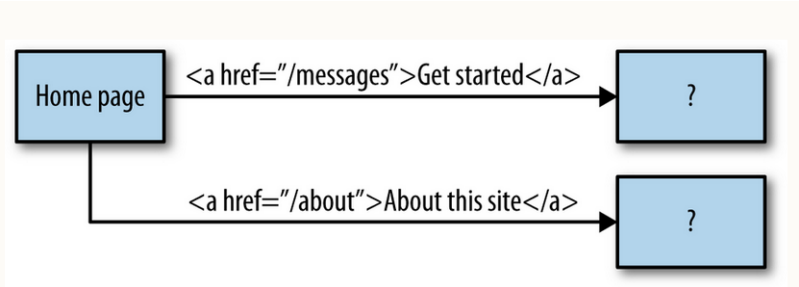
Le serveur ne connaît pas et n'a pas à connaître l'état de ses clients. Une connexion HTTP est brève et immédiatement refermée (HTTP 1, plus vrai en HTTP 2) après envoi de la réponse HTTP.

Il revient au client de gérer la complexité de communiquer leur état. Le serveur peut ainsi gérer plus de clients (pas besoin de garder des sessions ouvertes).

Compromis à faire sur l'authentification (JWT, cookie).

> Avec le cache, c'est ce qui permet au web de monter en charge et c'est ce qui a permis au web de survivre à sa popularité.

Interface uniforme : Autodescriptif



Quand on récupère une page web on a :

- Ce qu'on a demandé
- Une indication sur ce qu'il y a d'autres (ex: navbar)

L'état désiré d'une ressource doit être représenté dans le message envoyé au client :

- Le texte d'une balise `<a>` décrit la ressource derrière l'URL
- Un formulaire avec son contenu indique les états possibles de transition des ressources

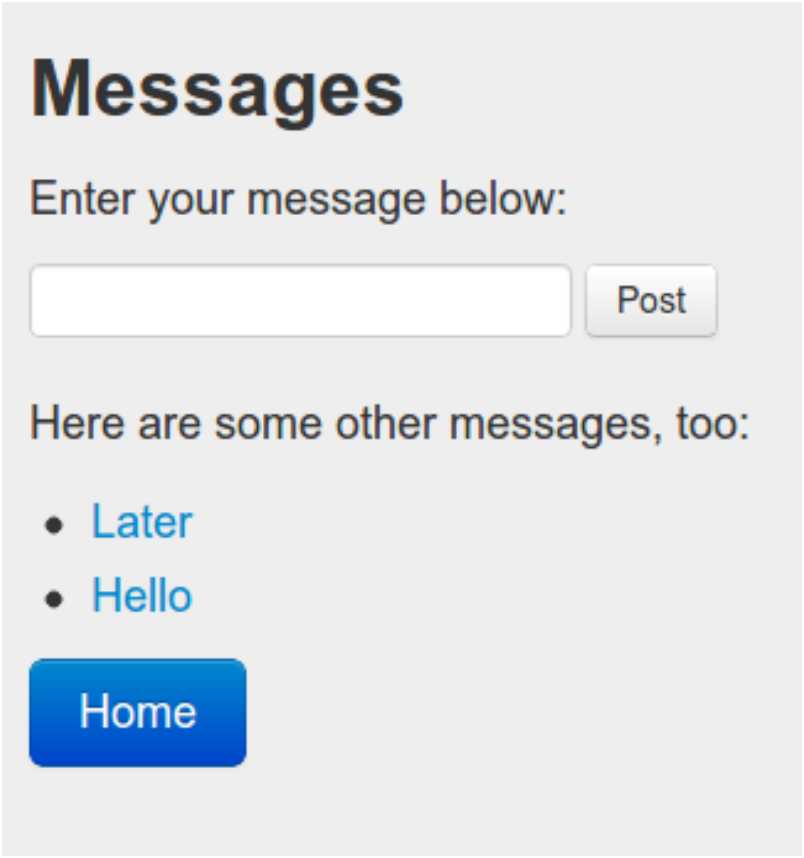
Alice continue sa navigation

```
GET /messages HTTP/1.1
Host: www.youtypeitwepostit.com
```

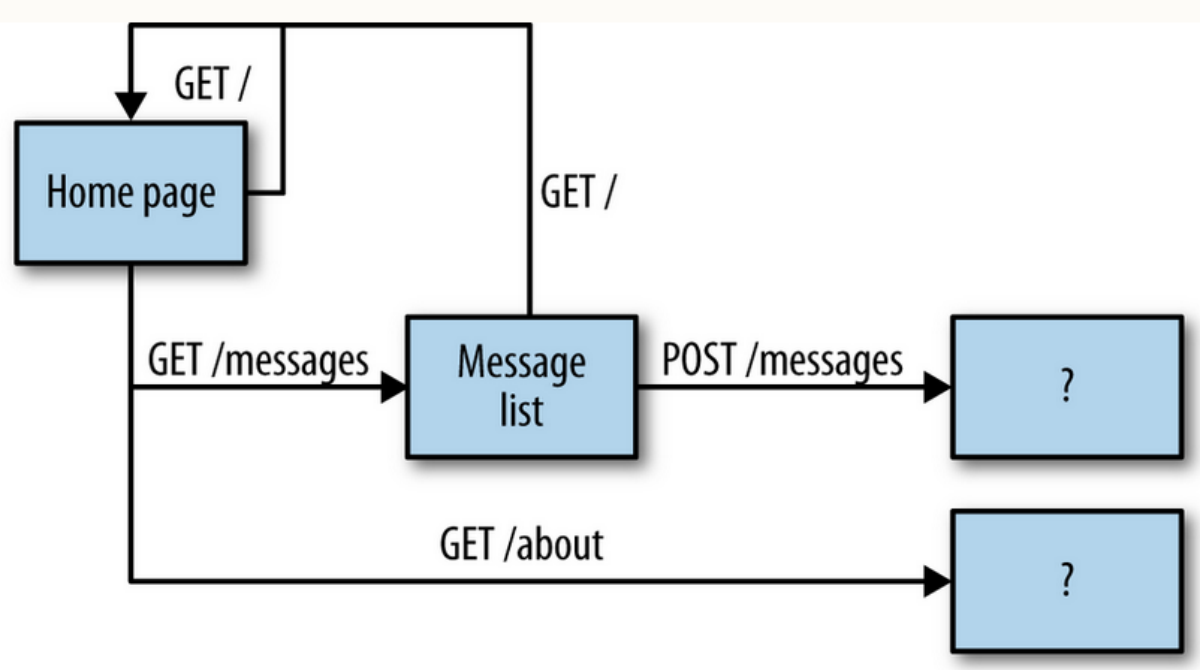
`GET` : HTTP Method pour dire "donne moi une représentations de cette ressource". GET est la méthode HTTP par défaut dans un navigateur web.

Réponse du serveur

```
HTTP/1.1 200 OK
Content-type: text/html
...

<!DOCTYPE html>
<html>
    <head>
        <title>Messages</title>
    </head>
    <body>
      <div>
        <h1>Messages</h1>

        <p>
          Enter your message below:
        </p>

        <form action="http://youtypeitwepostit.com/messages" method="post">
            <input type="text" name="message" value="" required="true"
                   maxlength="6"/>
            <input type="submit" value="Post" />
        </form>

        <div>
          <p>
            Here are some other messages, too:
          </p>
          <ul>
            <li><a href="/messages/32740753167308867">Later</a></li>
            <li><a href="/messages/7534227794967592">Hello</a></li>
          </ul>
        </div>

        <p class="links">
          <a href="http://youtypeitwepostit.com/">Home</a>
        </p>

      </div>
    </body>
</html>
```



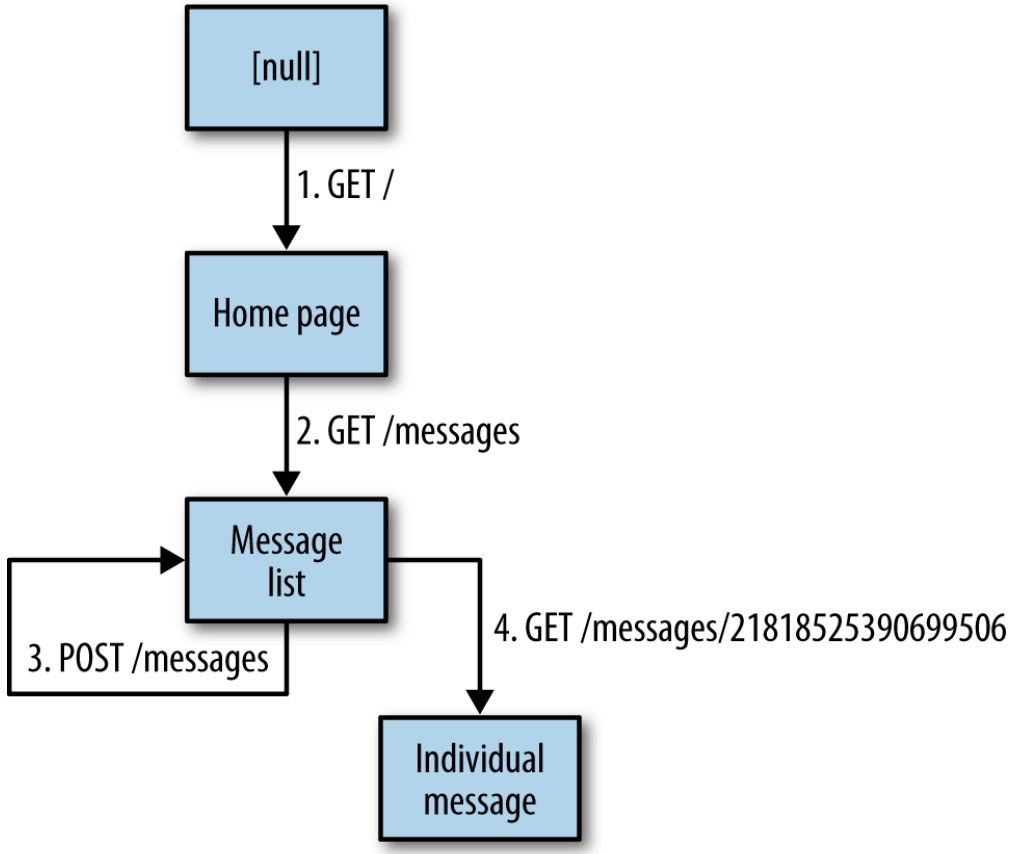
Carte actuelle du site web



Navigation du point de vue du client (Alice)

Du point de vue du client, on parle d'*application state* ("sur quelle page on est")

Quand on clique sur un lien, on passe d'un état de l'application à un autre (on change de page)



Navigation du point de vue du serveur

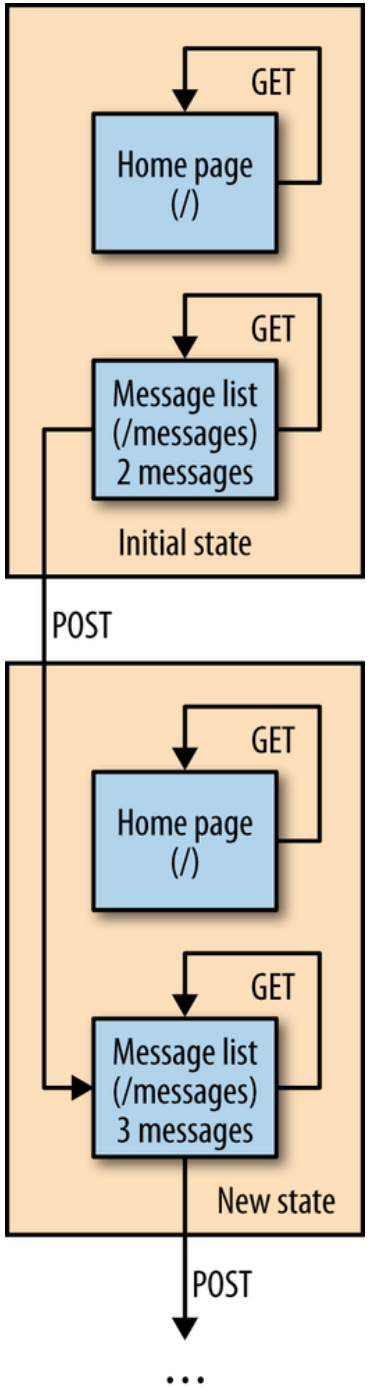
On parle de *resource state*.

Soumettre un formulaire *change* l'état de la **ressource** (un nouveau message est ajouté)

Le serveur *manipule* le client (représentation des états possibles). Le client manipule le serveur (envoie une représentation de l'état désiré des ressources que le serveur peut comprendre: URL, verbe HTTP, contenu de l'enveloppe)

> Le serveur indique au client les possibilités qu'il a pour modifier ses ressources. Le client manipule le serveur (post par exemple) pour modifier son état.

> REST: REpresentational State Transfer



Les autres contraintes (propriétés du web)

1. Client-serveur
2. Interface uniforme
    i. Verbes ou méthodes HTTP
    ii. Manipulation des ressources via des représentations
    iii. Hypermedia As The Engine Of the Application State* (HATEOAS)
3. Système en couches
4. Cache
5. Code à la demande

1. Client-serveur

Le web fonctionne sur l'architecture client-serveur. Chaque composant a un rôle précis :

- le **client** fournit au serveur une *représentation* de l'état désiré des ressources
- le **serveur** donne au client une *représentation* de l'état actuel des ressources et les moyens de les modifier

> Contrairement au protocole FTP par exemple, un client ne peut pas et n'accède jamais directement aux ressources. Un serveur web met à disposition des représentations de ses ressources et éventuellement des moyens pour les modifier.

2. Interface uniforme : verbes (ou méthodes) HTTP communes

Tous les accès aux ressources se font à travers l'interface uniforme du protocole HTTP. Cette interface comprend les 4 verbes (ou méthodes HTTP) de base : `GET`, `POST`, `PUT`, `DELETE` et les deux verbes auxiliaires `HEAD` et `OPTIONS`.

> **Ces 6 verbes sont à connaître** !

La complexité de votre application (la variété des ressources et les liens entre elles) doit être placée dans les représentations des ressources, pas au niveau des méthodes d'accès (comme dans l'architecture RPC).

> L'interface uniforme du web humain (proposé par les navigateurs) est limitée à `GET` (cliquer sur un lien) et `POST` (soumettre un formulaire). Toute votre navigation sur le web se résume à cette interface !

2. Interface uniforme : Manipulation des ressources *via* des *représentations*

- Le client ne manipule *jamais* les ressources, mais des *représentations* de ces ressources (documents HTML, JSON, XML, etc.).
- Le serveur ne donne *jamais directement accès à ses ressources*, mais à des représentations de celles-ci (un formulaire soumis par un client est une représentation d'un état désiré des ressources que le serveur peut comprendre et c'est lui qui effectuera la manipulation des ressources)

2. Interface uniforme : *Connectedness* ou *Hypermedia As The Engine Of the Application State* (HATEOAS)

Une représentation des ressources (*resource state*) inclut des liens vers les ressources connexes. Cela permet aux agents (humains ou non) de parcourir l'information et d'utiliser des programmes de manière logique et utile pour explorer le web.

> La présence ou l'absence d'un lien sur une page HTML est un fait important en ce qui concerne l'*application state.* La navbar d'une page HTML par exemple est un composant *HATEAOS*.

3. Système en couches

Permets à des composants intermédiaires sur le réseau (proxies, passerelles, reverse-proxy, etc.) d'être déployés de manière transparente entre un client et un serveur.

4. Cache

Contrainte la plus importante de l'architecture du web !

> C'est ce qui permet au web de monter en charge et c'est ce qui a permis au web de survivre à sa popularité.

Permets au serveur de déclarer la *cachabilité* des représentations de ses ressources.

Aide à

- réduire la latence vécue par le client
- augmenter la disponibilité d'un site web
- éviter requêtes inutiles (si les ressources n'ont pas changé entre deux requêtes)
- réduire le coût du web (attente, délais, transport, usage et sollicitation du réseau, consommation électrique, etc.)

Le cache peut être placé :

- côté client (navigateur)
- côté serveur (ex: mise en cache de requêtes SQL, de pages web dynamiques, templates, assets, etc.)
- sur le réseau (Content Delivery Network (CDN))

5. Code à la demande

Contrainte optionnelle. Permets au serveur de transférer temporairement l'exécution de code au client (scripts).

> Tends à créer un couplage technologique entre le serveur et ses clients (les navigateurs doivent être mis à jour, node.js !, etc.), car le client doit être capable de comprendre et d'exécuter le code qu'il a téléchargé depuis le serveur. Le serveur demande au client d'être capable d'exécuter son code pour modifier l'état des ressources !

Bénéfices des contraintes REST

- Le web a survécu à son succès grâce à ces contraintes (protocole HTTP, sans état et cache)
- Plus simple à comprendre, plus simple à utiliser
- Plus interopérable (avec d'autres systèmes web ou non)
- Plus facile à faire évoluer et à maintenir

> Le but d'une api RESTful est de bénéficier de tous les avantages procurés par le respect de ces contraintes côté applicatif (*in user land*)

Designer une API *sur* et *pour* le web (web API)

La manière (*comment*) d'utiliser

- les URI (URL)
- les méthodes HTTP
- le contenu de l'enveloppe HTTP
- la portée (*scoping information*)

définit l'**architecture** ou le design d'une web API.

> Tout revient *à l'usage* que l'on fait des contraintes et des technologies mises à disposition par le web.

Résumé et quiz

- Quiz
- Faire l'exercice 1

Différentes architectures d'API sur le web

- Architectures de type RPC (*Remote Call Procedure*), comme SOAP (ancien acronyme *Simple Object Access Protocol*). L'orienté objet appliqué au web
- Architecture de type ROA, (*Resource Oriented Architecture*), respectant les contraintes REST
- Systèmes hybrides REST-RPC
- API GraphQL
- etc.

Voyons cela en pratique, via un exemple.

Exemple d'architecture `RPC`, faire du *non REST* sur le web

> Exemple de code source d'un service XML-RPC.

Cette procédure *crée* une requête HTTP vers un système suivant l'architecture RPC pour trouver la description d'un produit par son code-barre

```ruby
#!/usr/bin/ruby -w
# xmlrpc-upc.rb

require 'xmlrpc/client'
def find_product(upc)
  server = XMLRPC::Client.new2('http://www.upcdatabase.com/rpc')
  begin
    response = server.call('lookupUPC', upc)
  rescue XMLRPC::FaultException => e
    puts "Error: "
    puts e.faultCode
    puts e.faultString
  end
end

# Appel de la méthode *remote*
puts find_product("001441000055")['description']
# Réponse: "Trader Joe's Thai Rice Noodles"
```

Exemple d'architecture `RPC`, faire du *non REST* sur le web

L'appel à `find_product` fabrique le document XML suivant

La procédure à appeler est les arguments sont placés dans un document XML, qui sera ensuite *placé dans l'enveloppe* de la requête HTTP

```
<?xml version="1.0" ?>
 <methodCall>
  <methodName>lookupUPC</methodName>
  <params>
   <param><value><string>001441000055</string></value></param>
  </params>
 </methodCall>
```

Exemple d'architecture `RPC`, faire du *non REST* sur le web

Requête HTTP générée

```
POST /rpc HTTP/1.1
Host: www.upcdatabase.com
User-Agent: XMLRPC::Client (Ruby 1.8.4)
Content-Type: text/xml; charset=utf-8
Content-Length: 158
Connection: keep-alive

<?xml version="1.0" ?>
<methodCall>
 <methodName>lookupUPC</methodName>
 ...
</methodCall>
```

- Le document XML est placé dans l'enveloppe HTTP
- Toujours la même URL (http://www.upcdatabase.com/rpc)
- Toujours la même méthode HTTP (`POST` en général)

Architecture RPC, résumé

- Le contenu du document XML change (en fonction de la procédure appelée)
- **L'enveloppe HTTP reste identique**:
  - même URI
  - même méthode HTTP
  - mêmes en-têtes
- Derrière l'URI: une application *qui sait* ouvrir et **transformer le contenu** des enveloppes (documents XML) **en appels de procédures**

Une API web de type RPC **ignore la plupart des fonctionnalités offertes par le protocole HTTP** (un seul URI, une seule méthode).

Une API web de type RPC **n'est pas RESTful**, c'est-à-dire qu'elle n'est pas bâtie exclusivement sur les contraintes de l'architecture web (couche logique supplémentaire)

Une API web orientée ressource (ou *RESTful*)

Une API web respectant les contraintes REST (i.e l'architecture du web) est dit *RESTful*.

Exemple précédent en RESTful

Une API web RESTful répondrait à autant d'URI que de valeurs différentes pour la portée

```
require 'open-uri'
upc_data = open('http://www.upcdatabase.com/upc/00598491').read()
...
```

- *Portée* contenue dans l'URI
- Un URI pour chaque code UPC
- **Nom de la procédure à appeler est définie par la méthode HTTP** (`GET`)

Les contraintes REST

Définie par Roy T. Fielding (principal auteur de la spécification HTTP, fondateur d'Apache) dans les années 2000

> Au cours de la procédure de standardisation de HTTP, on m'a appelé pour **défendre les choix d'architecture du Web**. C'est une tâche **extrêmement compliquée** dans la mesure où la procédure accepte les propositions de n'importe qui sur un sujet qui était en train de devenir rapidement le centre d'une industrie entière. Je recevais les commentaires de plus de 500 développeurs, dont de nombreux étaient des ingénieurs renommés avec des décennies d'expérience, et je devais tout expliquer, des notions les plus abstraites des interactions du Web jusqu'aux détails les plus subtils de la syntaxe de HTTP. **Cette procédure a réduit mon modèle à un ensemble fondamental de principes, propriétés et contraintes qui sont aujourd'hui appelés REST**. (Roy T. Fielding, 2006)

Une application *RESTful* dispose d'une architecture qui *embrasse* l'architecture du web et le protocole HTTP. Par exemple, un site statique est *par définition RESTful*.

API RESTful, un *service web* comme les autres

- Orienté *ressources* (on parle d'architecture *ROA*, Resource Oriented Architecture, application des contraintes REST au web)
- Portée véhiculée par l'URI
- **Doit fonctionner comme le web humain** (sous-ensemble du web):
  - Les représentations doivent être *navigables*
  - Procédure à appeler va *de paire* avec la méthode HTTP (utilisation de l'interface uniforme)
  - Etc.

> Comprendre *REST*, **et bâtir un service web disposant d'une API RESTful**, implique de bien comprendre l'architecture du web et le protocole HTTP.

Conception d'API *orientée ressource*

Une démarche générale, proposée par Leonard Richardson et Sam Ruby

1. **Déterminer** l'ensemble de données
2. **Décomposer** l'ensemble de données en ressource

> A mener en parallèle avec la conception de la base de données (dictionnaire des données, MCD)

Pour chaque type de ressource:

3. **Nommer** les ressources avec des URI
4. **Implémenter** un sous-ensemble de l'interface uniforme (GET, POST, DELETE, PUT)
5. **Étudier** la ou les représentations acceptées par les clients
6. **Concevoir** la ou les représentations à mettre à disposition des clients
7. **Intégrer** la ressource parmi celles qui existent déjà, en utilisant des hypermédias
8. **Envisager** la progression typique des évènements: qu'est-ce qui est censé se produire ? Le flux de contrôle standard comme le protocole de publication Atom peut aider.
9. **Considérer** les cas d'erreurs: qu'est-ce qui peut mal se passer ? Encore une fois, les flux de contrôle standard peuvent aider.

Exercice

Faire l'exercice 2

Exemple d'API RESTful bien conçue

- Documentation sur l’API REST GitHub

Design des *représentations* des ressources (aka, format des données échangées)

Différentes spécifications (pas de standard) :

- JSON API
- HAL
- JSON LD, soutenu par le W3C
- Hydra
- ...

Voir ce guide qui fait une bonne synthèse des différentes spécifications existantes.

Du bon usage des *hypermédias*

Un hypermédia fournit la possibilité de créer des liens entre ressources (documents, images, services, parties de document, etc.). HTML met par exemple à disposition des hypermédias, sous la forme de balises `<a>`, `<img>`, `<form>`, etc.

> Un hypermédia généralise le concept initial d'hypertexte.

Lire ce billet de blog de Roy Fiedling, qui insiste sur le fait qu'une API RESTful doit être orientée *hypertexte* (ou de manière générale *hypermédia*)

Design des représentations des ressources API RESTful: Spécification HAL

> HAL is a generic media type "application/hal+json" with which Web APIs can be developed and exposed as *series of links*. Clients of these APIs can select links by their link relation type and traverse them in order to progress through the application.

- La spécification HAL définit quelle structure générale donner à la représentation de la ressource et *comment mettre en place de manière standardisée les liens vers les ressources connexes* (connectivité de REST). Elle ne vous dit rien sur le reste (structures de données.)
- Renvoie une ressource avec le mime type `application/hal+json`, un document au format JSON.

Exemple d'une représentation (document JSON) conforme à la spécification HAL

```
#Exemple d'une réponse HTTP d'un service web conforme à la pécification HAL
GET /orders/523 HTTP/1.1
   Host: example.org
   Accept: application/hal+json

   HTTP/1.1 200 OK
   Content-Type: application/hal+json

   {
     "_links": {
       "self": { "href": "/orders/523" },
       "warehouse": { "href": "/warehouse/56" },
       "invoice": { "href": "/invoices/873" }
     },
     "currency": "USD",
     "status": "shipped",
     "total": 10.20
   }
```

HAL: *Resource Object*

La racine du document renvoyé par une API HAL doit être un Resource Object.

Un *Resource Object* est **une représentation d'une ressource au format JSON**, avec **deux propriétés réservées**:

- `_links`: contiens les liens vers les ressources connexes
- `_embedded`: contiens les ressources demandées sur l'URI

Toutes les autres propriétés doivent être du JSON valide, et *représenter l'état courant de la ressource*.

HAL, exemple

```
GET /orders HTTP/1.1
   Host: example.org
   Accept: application/hal+json

   HTTP/1.1 200 OK
   Content-Type: application/hal+json

   {
     "_links": {
       "self": { "href": "/orders" },
       "next": { "href": "/orders?page=2" },
       "find": { "href": "/orders{?id}", "templated": true }
     },
     "_embedded": {
       "orders": [{
           "_links": {
             "self": { "href": "/orders/123" },
             "basket": { "href": "/baskets/98712" },
             "customer": { "href": "/customers/7809" }
           },
           "total": 30.00,
           "currency": "USD",
           "status": "shipped",
         },{
           "_links": {
             "self": { "href": "/orders/124" },
             "basket": { "href": "/baskets/97213" },
             "customer": { "href": "/customers/12369" }
           },
           "total": 20.00,
           "currency": "USD",
           "status": "processing"
       }]
     },
     "currentlyProcessing": 14,
     "shippedToday": 20
   }
```

Idempotence de l'interface commune

> *Idempotence* : Propriété d'une opération, d'avoir le même effet qu'on l'applique une ou plusieurs fois. Par exemple, multiplier par 0 ou par 1.

- `GET`, `HEAD`, `OPTIONS` : méthodes *sûres*. Ne modifie aucune ressource. Effectuer plusieurs appels ne fait rien.
- `PUT`, `DELETE` : méthodes *idempotentes*. Effectuer plusieurs `PUT` vers un URI a (et doit avoir !) le même effet qu'un seul.
- `POST` : **ni sûre, ni idempotente !** Aucun moyen de savoir à l'avance ce que fera un `POST` sur l'état des ressources. A vous de le gérer correctement dans votre code applicatif.

API RESTful avec node.js

Voir la démo en cours sur la conception et l'implémentation d'une API de billetterie de concerts.

Utiliser le *starter-pack* mis à votre disposition à cette adresse.

Sécurité *côté client*: *Same Origin Policy* (SOP) et *CORS*

La *Same Origin Policy* est implémentée par tous les navigateurs dignes de confiance.

> Voir la démo en ligne commentée et les ressources associées.

Sécurité *côté serveur*: identification, authentification, autorisation (JWT) et autres aspects

- *identification*: le système s'assure qu'il vous connaît (*login*)
- *authentification*: le système s'assure que vous êtes bien la personne vous prétendez être (*password*, on peut renforcer cette phase avec de l'authentification à double, triple facteurs, biométrie, etc.)
- *autorisation*: le système s'assure que vous avez le droit d'effectuer telle action (consulter, modifier une ressource)

JSON Web Token

Une application RESTful n'a par définition pas d'état (pas de session par exemple). Le serveur ne garde aucune information sur le client une fois la réponse HTTP envoyée. Pour éviter d'avoir à vous authentifier *à chaque fois que vous demandez l'autorisation d'accéder à une ressource*, on peut utiliser le standard JSON Web Token.

En ce sens JSON Web Token est un document signé qui permet au serveur de vérifier que le client a l'autorisation d'accéder à une ressource (en lecture ou écriture) sans avoir à l'authentifier à chaque requête HTTP (passer le login et le mot de passe à chaque requête HTTP).

La manière dont est construit un JWT (grâce à la signature par clef secrète) permet de s'assurer que le document JSON émis par le serveur ne peut pas être modifié par le client *sans devenir automatiquement invalide*. Un JWT est donc avant tout un document qui peut être échangé entre deux parties (un client et un serveur) garantissant son intégrité.

Lorsqu'un JWT est utilisé pour autoriser un client à accéder à une ressource, on parle d'*access token*.

JSON Web Token

- Le serveur identifie et authentifie le client
- Il fabrique une chaîne de caractère signée avec une clef secrète (une chaîne de caractère conservée secrètement côté serveur): c'est le JSON Web Token. Il est constitué de 3 parties, séparés par des `.`:
  - Un header: contient l'algorithme de hachage utilisé pour encoder le JWT et le type (JWT)
  - Un corps, le *payload*: contient les informations utiles à l'application web (par exemple stocker un rôle)
  - Une *signature*: la signature est le résultat du hachage du header, du payload (encodés en base 64) et du secret. C'est l'élément qui assure l'intégrité du token émis.
  - Un exemple de JWT : `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c`
- Le serveur retourne le JWT au client (dans le navigateur il peut être placé dans un cookie ou le localStorage)
- Lors de la prochaine requête HTTP nécessitant une autorisation, le client envoie le JWT avec sa requête
- Le serveur vérifie que le JWT est présent. Il le redécoupe en 3 parties et récupère (et décode) le header et le payload.
- À l'aide de son secret, il **régénère** la signature puis la compare avec la signature reçue par le client :
  - Si elle correspond à la signature envoyée par le client, la requête est acceptée
  - Si elle ne correspond, le JWT est invalide (par exemple modifié par le client), la requête est rejetée

JSON Web Token : démos

- Démo en *vanilla* PHP
- Démo node.js avec express.js et le paquet jsonwebtoken

JSON Web Token

- Un JWT est signé côté serveur par une clef qui doit demeurer secrète
- JWT permet d'authentifier et donc donner une permission à un client sans avoir à stocker d'état (le secret n'est pas un état dépendant du client)
- Attention, JWT ne protège pas
  - Contre le vol du token : si on vous dérobe votre token, c'est comme si on vous volait votre mot de passe
  - Contre les attaques CSRF !

(Quelques) bonnes pratiques:

- Ne surtout pas placer de données sensibles dans un token (comme un mot de passe) ! N'importe qui peut décoder votre token sans trop d'efforts !
- Faire expirer le token au bout d'un certain moment, pour diminuer les risques de vol de token ou de craquage
- Avoir un secret fort
- Utiliser un bon algorithme de hachage (RS256,EdDSA, etc.), surtout pas SHA1 ou MD5.
- Assurez-vous que l'authentification (envoi du mot de passe et login) passe par une connexion sécurisée (TLS/SSL)
- Si votre secret a fuité, changez-le immédiatement !

Protéger votre API

Pour protéger votre API, il faudra compter sur certaines mesures en fonction de vos besoins et des ressources exposées :

- Politique CORS (*Cross Origin Resource Sharing*), attention contrainte seulement appliquée aux clients navigateurs webs (web humain). Voir la démo SOP pour en apprendre plus.
- Authentification
- Autorisations (JWT)
- Rate Limitation (limiter le nombre de requêtes HTTP dans un temps donné pour un agent)
- Captchas
- *White* list / *Black* list
- Etc.

Ce qu'il faut retenir

- REST est un ensemble de contraintes qui servent de fondements à l'architecture du web. Une API RESTful est une API sur le web qui adhère complètement aux contraintes du web (sans-état, connectivité, adressabilité, etc.)
- Pour designer une API RESTful (ou *orienté ressources*)
  - **4 concepts** :
    - Les ressources
    - Les noms des ressources (URI)
    - Leurs représentations (documents et leur format entre client et serveur)
    - Les liens entre elles
  - **4 propriétés** :
    - Adressabilité
    - Absence d'états
    - Connectivité (HYATEOS)
    - Interface uniforme

Références

Livres

- RESTful Web APIs, de Leonard Richardson, Mike Amundsen, Sam Ruby, O'Reilly, 2013. **S'il y a un livre à lire/étudier/feuilleter/avoir c'est celui-ci, must have**
- REST API Design Rulebook, de Mark Masse, O'Reilly, 2011
- RESTful Web Services, de Leonard Richardson, Sam Ruby, O'Reilly, 2007
- REST API Development with Node.js : Manage and Understand the Full Capabilities of Successful REST Development, 2nd Edition, de Fernando Doglio, Apress, 2018
- Bien architecturer une application REST, par Olivier Gutknecht, avec la contribution de Jean Zundel, Eyrolles, 2009

Sur le web

- HTTP Caching, une synthèse sur l'implémentation du cache du protocole HTTP. Attention, tous les navigateurs n'implémentent pas le standard au même point.
- Un tutoriel sur la mise en cache du protocole HTTP, un très bon tutoriel en français sur la mise en cache du protocole HTTP
- REST, une synthèse de David Gayerie sur REST de grande qualité, avec une bibliographie utile
- Welcome to the REST CookBook, portail sur différents aspects de REST
- How to get a cup of coffe (in REST)
- RFC3986: Uniform Resource Identifier (URI): Generic Syntax, T. Berners-Lee (W3C/MIT), R. Fielding (Day Software), L. Masinter (Adobe Systems)
- RESTful Web APIs: examples, Node.js code for the clients and servers used as examples in O'Reilly's "RESTful Web APIs".
- Same-origin policy: The core of web security @ OWASP Wellington, excellente présentation de Kirk Jackson de la Same Origin Policy avec démonstrations à l'appui. **À regarder**.
- REST APIs must be hypertext-driven, billet de blog de Roy T. Fiedling très intéressant sur le fait qu'une API RESTful doit être orientée *hypertexte* (ou de manière générale par les *hypermédia*)

Sur le web

- JSON Hypertext Application Language draft-kelly-json-hal-08, HAL representation pour les modèles de données. Une proposition de standard
- API RESTful, spécification des schémas de données *HAL*, les différents types d'hypermédia définis pour le protocole HTTP et pour construire des API plus robustes. Le livre de l'auteur Building Hypermedia APIs with HTML5 and Node, Amundsen, a l'air très intéressant
- API RESTful, spécification des schémas de données *JSON-LD 1.1*, *A JSON-based Serialization for Linked Data*, une autre spécification des données renvoyées par une API, soutenue et recommandée par le W3C
- Schema.org, *Schema.org is a collaborative, community activity with a mission to create, maintain, and promote schemas for structured data on the Internet*. Propose une liste de schémas à suivre pour différents modèles de données.
- Microformats wiki, un wiki qui décrit des spécifications de structure de données interopérables
- Hydra, Hydra is an effort to simplify the development of interoperable, hypermedia-driven Web APIs. The two fundamental building blocks of Hydra are JSON-LD and the Hydra Core Vocabulary.
- Zalando RESTful API and Event Guidelines

Sur le web

- OpenAPI Specification, *The OpenAPI Specification (OAS)* defines a standard, programming language-agnostic interface description for HTTP APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. C'est la spécification que suit l'outil Swagger
- W3C: Cool URIs don't change, par Tim Berners-Lee
- W3C: Architecture of the World Wide Web, Volume One, écrit par le *W3C Technical Architecture Group*: Tim Berners-Lee (co-Chair, W3C), Tim Bray (Antarctica Systems), Dan Connolly (W3C), Paul Cotton (Microsoft Corporation), Roy Fielding (Day Software), Mario Jeckle (Daimler Chrysler), Chris Lilley (W3C), Noah Mendelsohn (IBM), David Orchard (BEA Systems), Norman Walsh (Sun Microsystems), and Stuart Williams (co-Chair, Hewlett-Packard). Une *bible* sur l'architecture du web.