Talent HUB

REST et Web API’s

01.06.2024

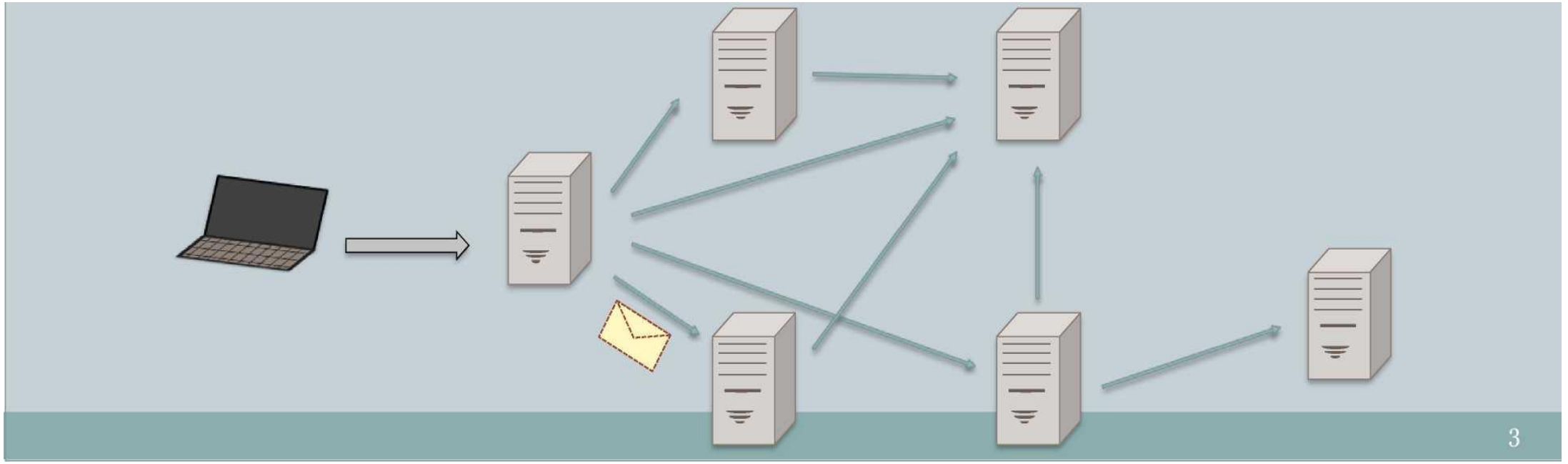
Agenda

- Introduction
- REST en théorie
- REST en pratique
- Web APIs
- Conclusion
- Questions

Introduction

Avant : les Services Web

- Échange de **services** en Machine-to-Machine (M2M)
- Notion d'**Architectures Orientées-Services** (AOS)
- Notions d'« orchestration » et de « chorégraphie » de services
- Ensemble de technologies (SOAP, WSDL, UDDI, BPEL...)



Introduction

Caractéristiques d'un service Web

Permet l'échange de messages(documents) et non de pages Web

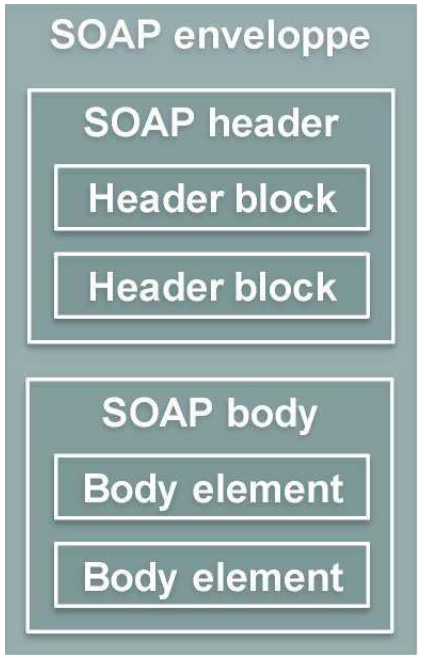
- XML, JSON…

Expose une API (Clients AJAX, autres services)

- Ensemble d'**opérations** disponibles (verbes)
  Ex : login, addUserToGroup, createReservation…

Utilise (ou pas) HTTP comme protocole de transport

- HTTP
  - Uniquement des requêtes POST
  - Beaucoup de «redites»
- Autres : SMTP, TCP, UDP, JMS…



Introduction

Les services Web : au final

Un mécanisme très puissant pour les applications Web complexes

Pas si «simple»

Redéploie toute une stack au-dessus de HTTP

Pas si interopérable

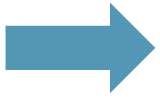
Problèmes d'encodage, de (dé)sérialisation des messages dus aux implémentations des outils

Pas si standard / réutilisable

L'API d'un service dépend des choix de conception
Couplage fort entre fournisseur (serveur) et consommateur (client)

Pas (du tout ?) scalable

Pas de cache, mauvaises performances dues à la stack et à la bande passante importante

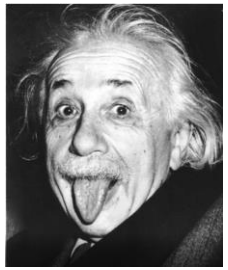


Perte des «bonnes propriétés» qui ont fait le succès du Web

REST en théorie

« La théorie, c 'est quand on sait tout et que rien ne fonctionne...

Albert Einstein



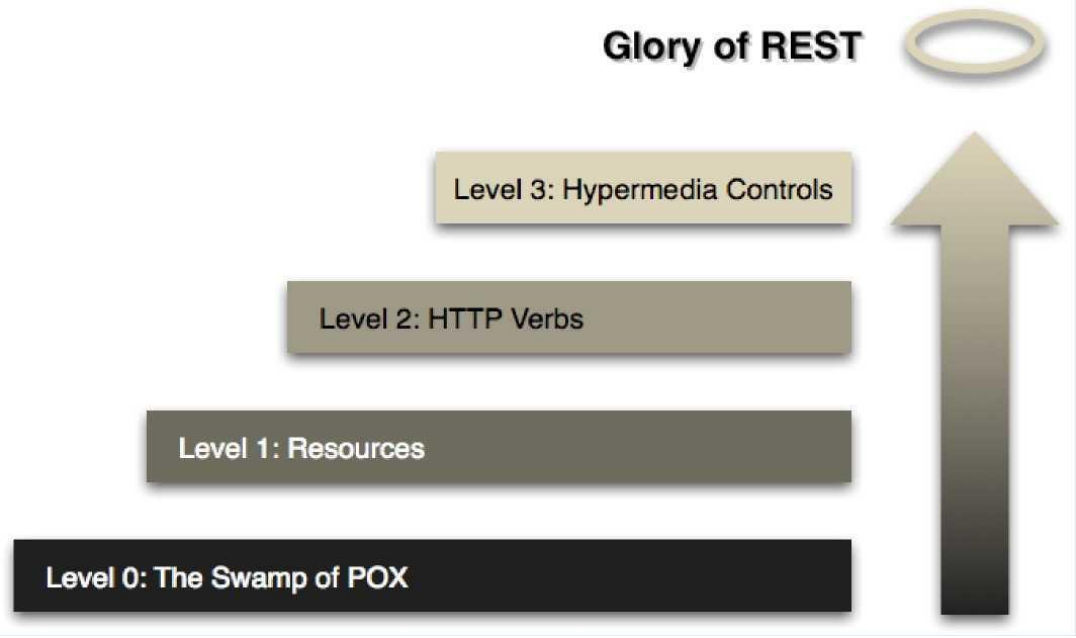
Selon toute vraisemblance, ce portrait d'Albert Einstein serait en fait le résultat d'un étonnant concours de circonstances. Le 14 mars 1951, le physicien allemande fête en effet son 72ème anniversaire, à côté de l'université de Princeton, dans l'État du New Jersey.
Suivi durant toute cette journée par une cohorte de journalistes, le vieil homme est mitraillé de flashs provenant de nombreux appareils photos. Alors, par lassitude, celui qui voulait simplement fêter son anniversaire en compagnie de ses proches se retourne vers un des photographes... et tire ostensiblement la langue. Pris sur le vif, par un employé de l'agence de presse américaine « United Press International », le cliché est rapidement transmis à de nombreux journaux

REST en théorie

Modèle de maturité de Richardson

Largement tiré du billet de blog de Martin Fowler.

Le **Modèle de Maturité de Richardson** permet de réduire, niveau par niveau, le couplage entre un client et un serveur REST en utilisant simplement et strictement ce qui fait le cœur du Web : les URIs, les verbes (méthodes) et codes retour HTTP, les liens (hypermédias).
Atteindre le niveau 3 du MMR est la **garantie** d'avoir d'ores et déjà un système d'information **HATEOAS**, réellement Restful.



Source : https://architecturelogicielle.wordpress.com/2013/04/25/le-modele-de-maturite-de-richardson/

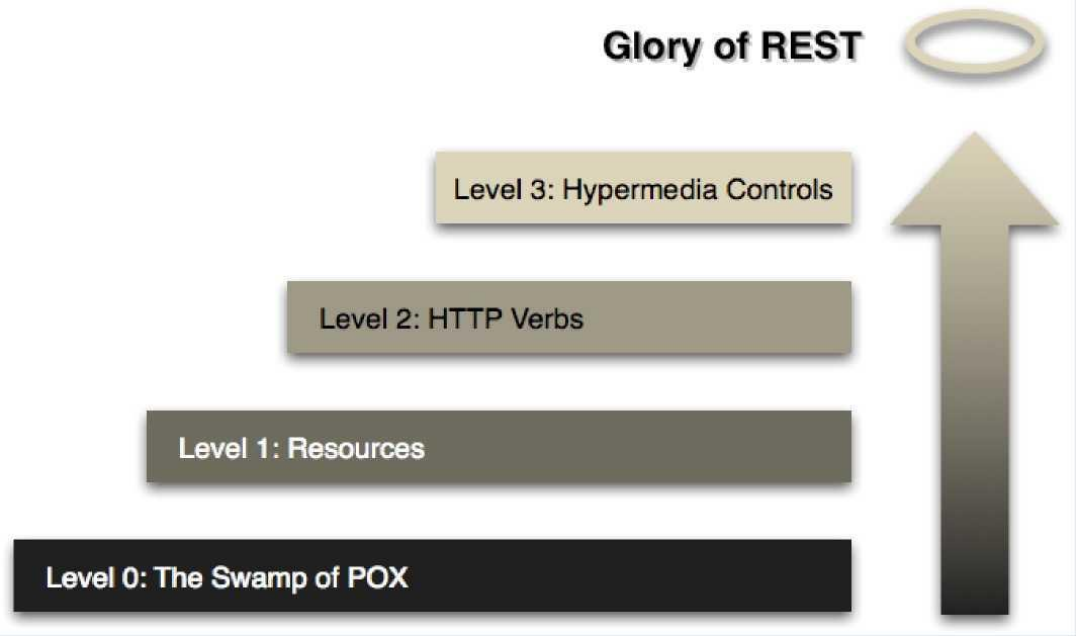
REST en théorie

Modèle de maturité de Richardson

3 principes fondateurs du Web ont été soulignés :

- les URIs
- le protocole HTTP
- les hypermédias

Ces trois principes constituent précisément les trois niveaux de maturité du modèle de Leonard Richardson, le niveau 3 représentant l'objectif à atteindre pour parler d'un système d'information distribué réellement REST, ou Restful.



REST en théorie

Style architectural

- Notion proposée par Roy Fielding dans sa thèse (2000) : *Architectural Styles and the Design of Network-based Software Architectures*.
- Défini comme un ensemble de **contraintes** imposées sur la conception d'un système pour garantir un certain nombre de **propriétés**.
- La question sous-jacente était d'identifier les contraintes à respecter dans le développement du Web :
  - pour conserver les propriétés ayant fait son succès,
  - pour corriger les problèmes constatés,
  - pour évaluer les évolutions futures des technologies.

https://opikanoba.org/tr/fielding/rest/

https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

REST en théorie

Objectifs

« REST emphasizes

- scalability of component interactions,
- generality of interfaces,
- independent deployment of components, and
- intermediary components to
  - ✓ reduce interaction latency,
  - ✓ enforce security, and
  - ✓ encapsulate legacy systems. »

(Roy Fielding)

REST en théorie

Objectifs - Passage à l'échelle

Problème de performance

La dimension et l'expansion constante du Web obligent à considérer dès le départ les problèmes d'échelle.

Problème de robustesse

Le système ne peut pas être 100% opérationnel 100% du temps.

REST en théorie

Objectifs - Passage à l'échelle

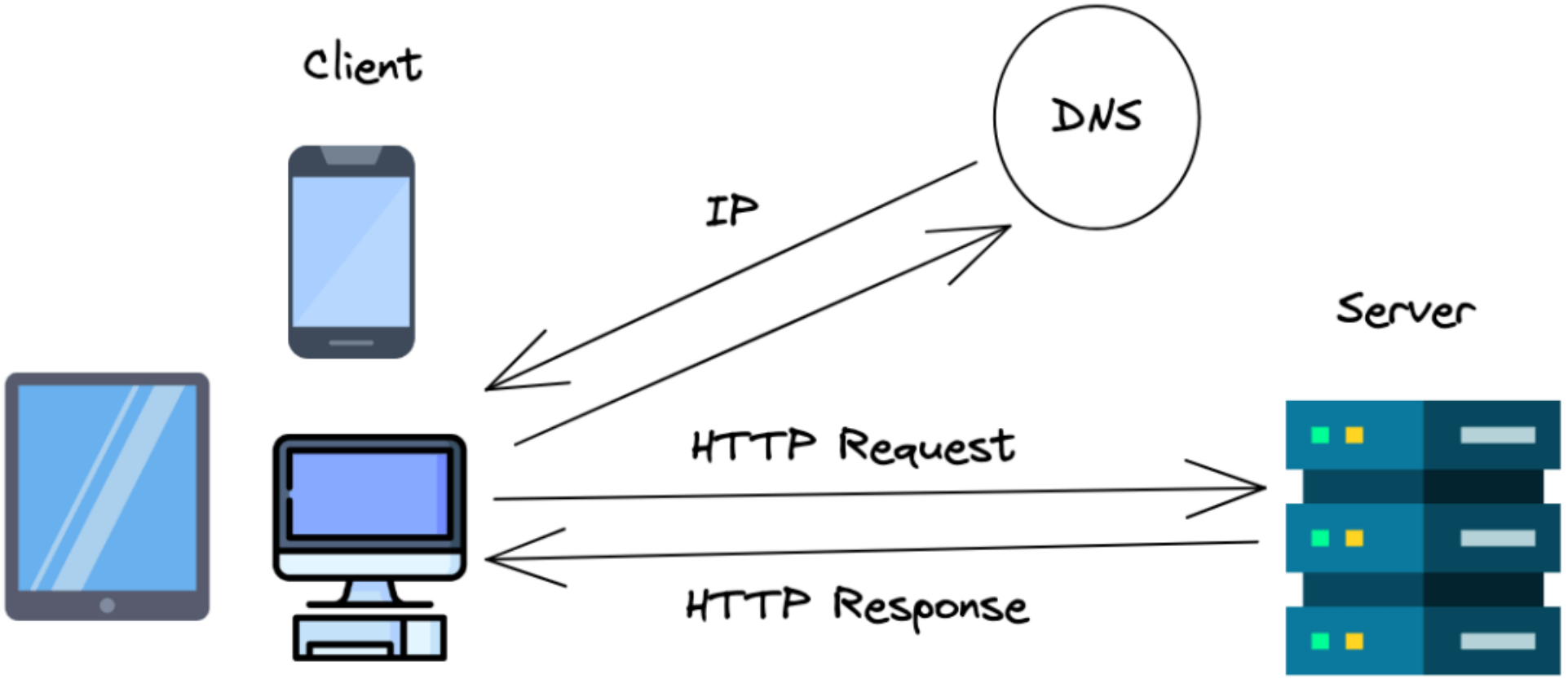
Rappel : Le Web 1.0

HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol : Verbes, codes de retour, media types, cache
URL	uniform resource locator : Ressource

REST en théorie

Objectifs - Passage à l'échelle

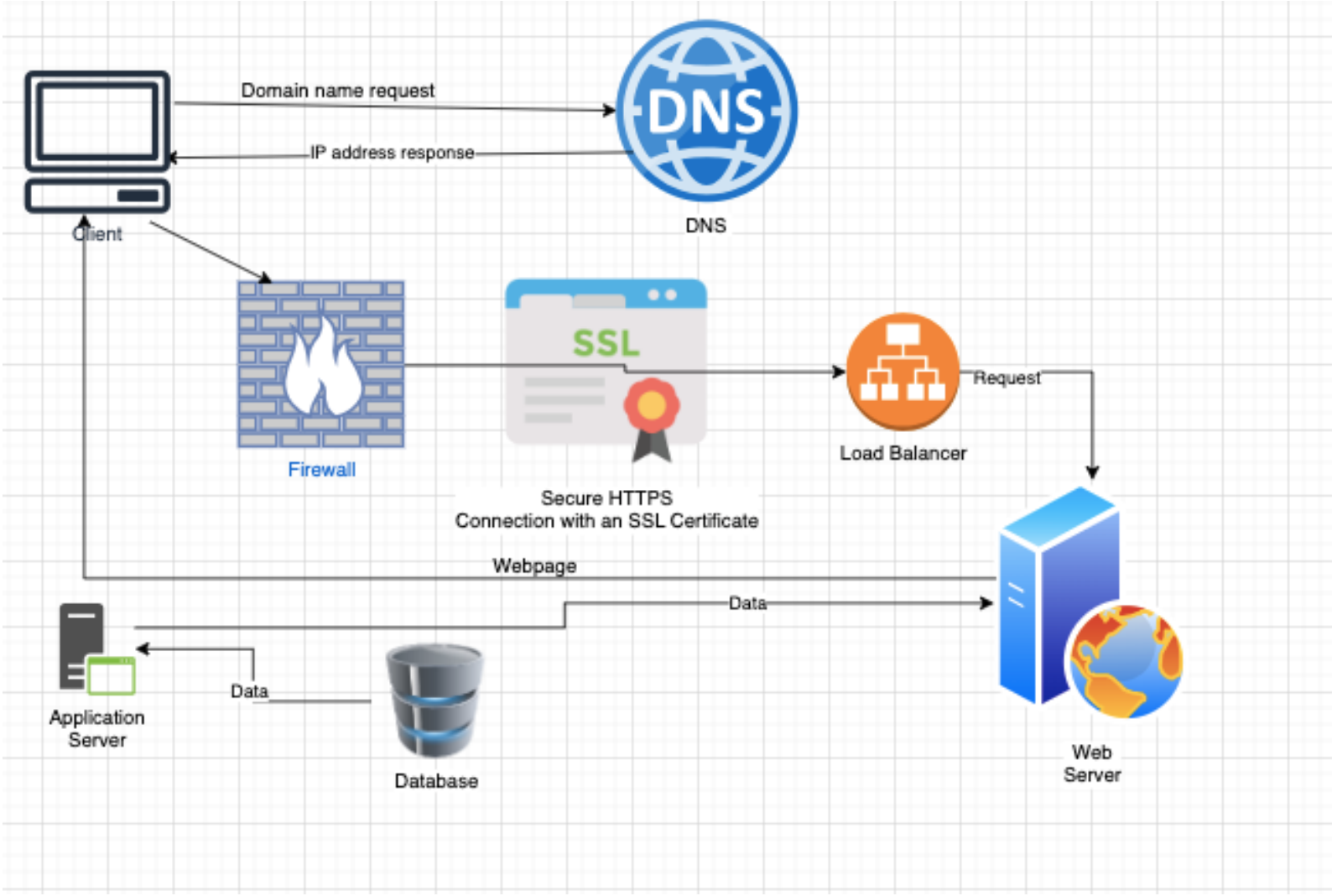
Rappel : la transaction HTTP



REST en théorie

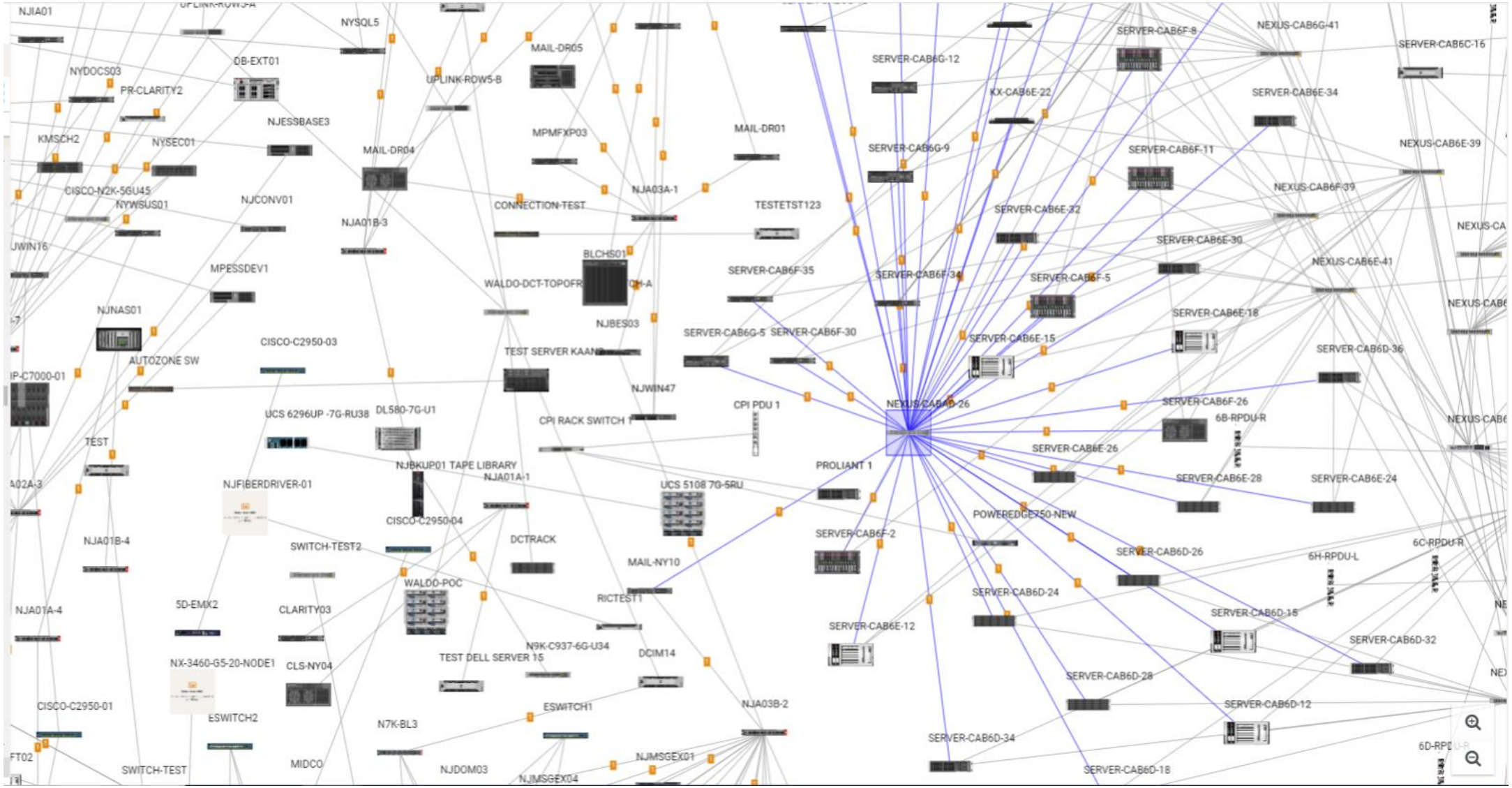
Objectifs - Passage à l'échelle

En vrai



REST en théorie

Objectifs - Passage à l'échelle



REST en théorie

Objectifs - Généralité des interfaces

Facilite le développement des composants

Réutilisation de bibliothèques standard, interopérabilité.

Évolutivité

Web de documents, Web de services, Web 2.0, Web des données, Web sémantique...

Principle of partial understanding (Michael Hausenblas)

REST en théorie

Objectifs - Déploiement indépendant des composants

Contrainte d'échelle

Pas de centralisation possible sur le Web.

Adoption et évolution rapides

Grassroot movement (poussé par le bas), « sélection naturelle », modèle du Bazar (Eric Raymond)

REST en théorie

Objectifs - Composants intermédiaires

Réduire la latence

en conservant en cache le résultat de certaines requêtes (proxy),
en répartissant la charge entre plusieurs serveurs redondants (reverse proxy).

Assurer la sécurité

en masquant l'accès à certains serveurs (proxy filtrant, reverse proxy, n-tiers).

Encapsulant des services

en adaptant les requêtes (gateway) ← généralité des interfaces

REST en théorie

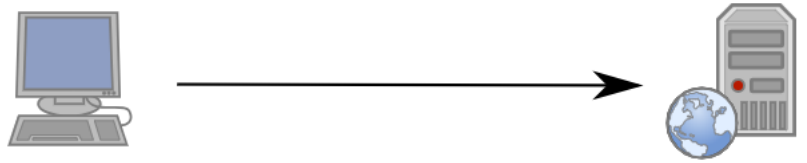
Contraintes

Contraintes de REST, d'après Fielding :

- client-serveur
- connexion sans état (*stateless*)
- support des caches
- interface uniforme
- système en couches
- code à la demande (optionel)

REST en théorie

Contraintes : Client-serveur



Separation of concern (Edsger W. Dikstra)

- Le serveur est responsable du stockage et de la cohérence des données (état des ressources).
- Le client est responsable :
  - ✓ de la présentation/du traitement des données,
  - ✓ de maintenir le contexte/état de l'interaction (cf. ci après).

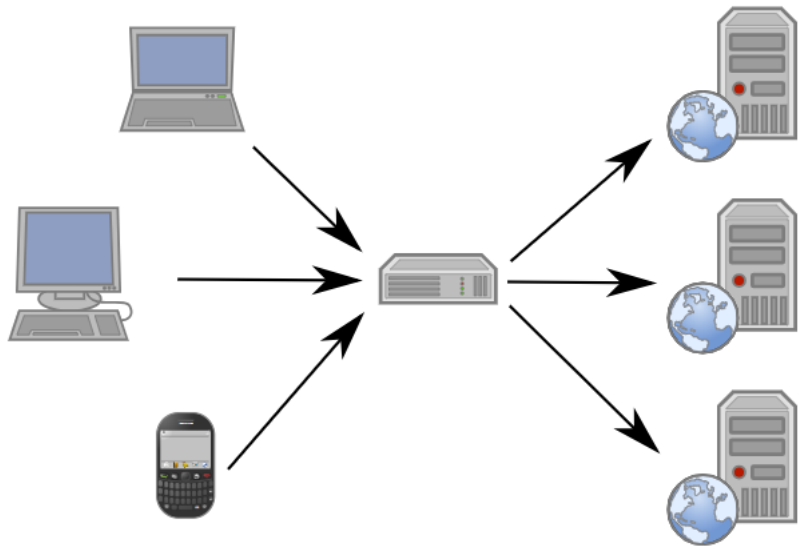
Remarques :

- Cette séparation peut être appliquée, même si le client et le serveur sont sur le même machine.
- Certains composants sont à la fois client et serveur (e.g. proxy). Ils respectent cependant la séparation des préoccupations vis-à-vis des composants avec lesquels ils communiquent.

REST en théorie

Contraintes : Système en couches

- Cette contrainte spécifie qu'un composant ne doit se soucier que de ses interlocuteurs directs.
- Le protocole HTTP assure lui même la cohérence des messages le long de la chaîne d'intérmédiaires en spécificiant, en fonction de leur sémantique, quelles parties d'une requête ou d'une réponse sont Hop-by-hop ou End-by-end.
- Permet la mise en place de Composants intermédiaires.

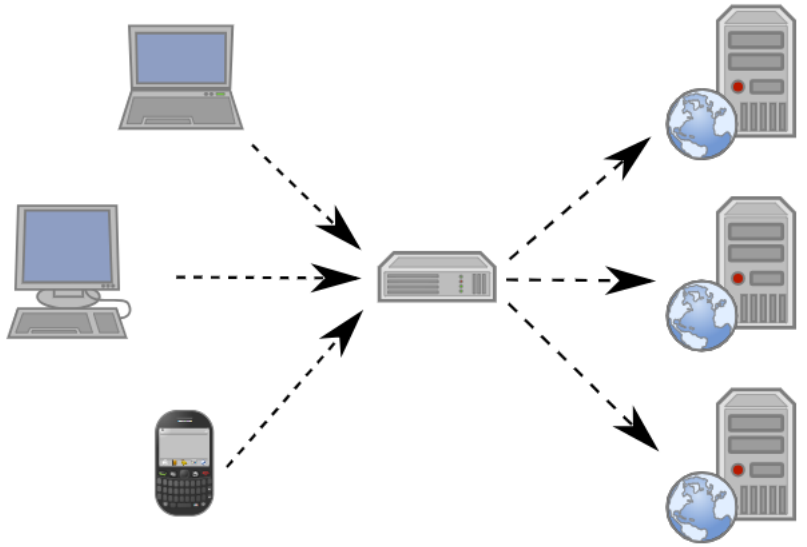


REST en théorie

Contraintes : Support des caches

Les verbes et les statuts pré-définis par HTTP ont une sémantique suffisamment précise pour informer les caches de la possibilité de « cacher » ou non un résultat.
NB: l'utilisation systématique de POST empêche l'exploitation des caches.

HTTP permet de contrôler plus finement le cache avec un certain nombres de champs d'en-tête (Cache-Control, Expires)



REST en théorie

Contraintes : Connexion sans état

Le serveur ne doit pas s'encombrer du *contexte* de l'interaction. Ce contexte est rappelé à chaque requête par le client, et les modifications de contexte sont indiquées dans la réponse.
→ auto-suffisance des messages.

Inconvénients

- surcoût en bande passante
- authentification à chaque requête (→ HTTPS)

Avantages

- performances (un serveur / beaucoup de clients)
- robustesse (redémarrage ou changement de serveur, **clients nomades**)
- facilite le travail des intermédiaires

REST en théorie

Contraintes : Interface uniforme (1) : ressources

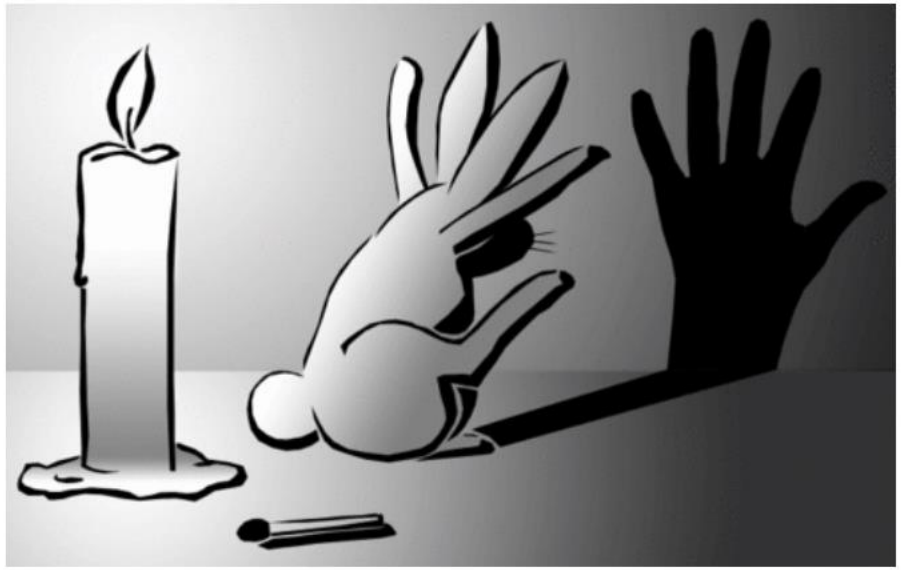
- Une ressource est un constituant du service (médecin, créneau horaire, rendez-vous, patient).
  - ✓ Une collection de ressource est aussi une ressource.
- Toute ressource est identifiée par une URL.
  - ✓ Réciproquement, deux URLs différentes identifient (a priori) des ressources différentes.
- Une ressource a un état interne (données stockées sur le serveur)
  - ✓ qui varie dans le temps,
  - ✓ et qui est inaccessible aux clients.

REST en théorie

Contraintes : Interface uniforme (2) : représentation

Une ressource n'est manipulée par le client qu'à travers des représentations de son état.

Tableau de René Magritte - Source Wikipedia



La représentation n'est pas nécessairement *exhaustive*.

REST en théorie

Contraintes : Interface uniforme (3) : messages auto-suffisants

La sémantique de la requête est entièrement portée par le message :

- verbe (GET, PUT, POST, DELETE)
- URL cible
- en-têtes éventuels
- représentation éventuelle

Très lié à Système en couches et Connexion sans état.

REST en théorie

Contraintes : Interface uniforme (4) : hypermédia

- Hypertext As The Engine Of Application State (HATEOAS)
- Notion d'affordance
- La représentation d'une ressource comporte les liens vers d'autres ressources (identifiées par leurs URIs).
- La sémantique du lien dépend du format de la représentation.
- L'état d'un client change en suivant les liens découvert dans les représentations.

NB : Une alternative consiste à construire un URI en fonction des informations fournies par une représentation. C'est le cas des formulaires HTML utilisant la méthode GET, e.g.:`http://www.google.fr/search?q=hypertexte`

REST en théorie

Contraintes : Code à la demande (optionnel)

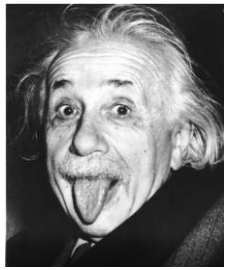
- Cette contrainte consiste à permettre au serveur d'envoyer au client non seulement la description d'un état, mais également une logique de traitement (programme).
- On peut voir AJAX (Asynchronous Javascript and XML) et ses dérivés (JSON) comme une généralisation de ce principe (même si AJAX n'utilise pas forcément des échanges RESTful).

REST en pratique

REST, concrétement

« *La théorie, c 'est quand on sait tout et que rien ne fonctionne…*
La pratique, c 'est quand tout fonctionne et que personne ne sait pourquoi…

Albert Einstein



REST en pratique

Approche orientée-ressource

En REST, la responsabilité du serveur est de gérer et d'exposer des **ressources**

…Et c'est tout (pas l'interaction avec le client).

Les ressources

- Symbolisent des objets du domaine
  - Peuvent correspondre à des entités en base ou être virtuelles
  - Sont identifiées par des noms (pas des verbes) (Plus proche de l'orienté-objet)
- Sont exposées par leurs **URL**
- Sont échangées sous forme de **représentations**(éventuellement partielles)

REST en pratique

Approche orientée-ressource

Travailler en orienté-ressource

1. **Concevoir un modèle de ressources arborescent**
   - Deux sortes de ressources
     - Instance (objet métier)
     - Collection d'instances
   - Les ressources «portent» les cas d'utilisation
     - Patterns GRASP
   - Les cas d'utilisations de l'application se rapportent à l'**état** interne de ces ressources
     - Opérations similaires à du CRUD
     - Opérations métier

Remarque : l'état interne des ressources n'est pas accessible au client

2 . Définir les URL qui permettent d'accéder aux ressources

Théoriquement, en REST, la forme des URL est libre si elles identifient une ressource de façon unique (d'où le nom)

Exemples

https://talehub.dev/search

https:// talehub.dev /userTH0098

Bonnes pratiques

- Faire apparaître le type et le nom de la ressource adressée
- Refléter la hiérarchie de ressources

https:// talehub.dev /users/userTH0098

REST en pratique

Approche orientée-ressource

Travailler en orienté-ressource

3. Spécifier les transferts de représentations d'états des ressources entre le client et le serveur pour chaque cas d'utilisation de l'application

- **Identifier l'ensemble des propriétés qui décrivent utilement l'état de la ressource**
  ⇒ souvent sérialisés sous forme de paires clés valeurs
- **Déterminer un/des format(s) de données pour les transactions HTTP**
  => Rappel : en MVC, une vue est une représentation des données, quel qu'en soit le format

REST en pratique

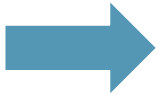
Transactions sans état

En REST, le client a la responsabilité
de gérer le contexte de l'interaction avec le serveur

…Le serveur ne conserve aucune trace des interactions passées

La connexion est dite **sans état** (stateless)

C'est le client qui maintient ses « variables de session »
=> Il transmet dans la requête le contexte nécessaire pour la traiter
En éliminant la gestion des sessions par le serveur
=> on élimine un goulot d'étranglement
=> on économise des ressources



Gain en performances et en scalabilité
Maintenance et évolution facilitées

REST en pratique

fa

Transactions sans état

Travailler en « Stateless

1. Identifier les éléments contextuels dont le serveur a besoin pour fonctionner
   - qui seraient des attributs de session en «stateful»
   - différents des paramètres de la requête (ne changent pas à chaque requête)
   - différents des identifiants des ressources
   - différents des données de configuration de l'application

2. Passer le contexte dans la requête
   - En paramètres : en fonction de la méthode HTTP
     Dans l'URL
     https://www.qwant.com/?q=test&t=web
     https://api.qwant.com/api/search/web?count=10&q=test&t=web&device=tablet&safesearch=1&locale=fr_FR&uiv=4
     Dans le corps de la requête
     {"query":"test","language":"francais","pl":"extff","lui":"francais cat":"web sc Cgfw2mIu1l4S20"}
   - Dans les headers HTTP
     Authorization
     Cookies (stateless , mais pas restful

REST en pratique

fa

Transactions sans état

Travailler en « Stateless

3. Récupérer le contexte côté serveur
    - Authentification, autorisation
        ⇒ Filtres
    - Informations nécessaires au traitement de la requête
        ⇒ Contrôleur
        ⇒ Quel type de MVC choisir ? **Pull based** ou **push based ?**

4. Traiter la requête en interrogeant la bonne instance du modèle

REST en pratique

Forme des URL

Bonnes pratiques

- Version : à la racine de l'arbre (pas nécessairement du serveur)
- Chemins
  - Utiliser des noms pluriels pour les collections
  - Utiliser des id d'instances comme sous-ressources des collections
  - Séparer les éléments par des slashs («/»)
  - Pas de slash final
- Paramètres : filtres, tri, pagination

  Exemples
  - https://talenthub.dev/api/v3/users/toto/friends/1
  - https://talenthub.dev /api/v3/messages?author=toto&sort=+title,-index
  - https://talenthub.dev /api/v3/messages?offset=100&limit=25

- URL pour les requêtes de CU «opérationnels»
  - Utiliser un verbe
  - Raccrocher le CU à l'URL de la ressource à laquelle se rapporte le CU

    Exemples
    - https://talenthub.dev /users/login
    - https://talenthub.dev /users/toto/playlist/play

REST en pratique

Utilisation avancée de HTTP

Historique des spécifications IETF

Juin 1999

RFC 2616: contient tout HTTP 1.1

Juin 2014

RFC 7230: Message Syntaxand Routing
RFC 7231: Semanticsand Content
RFC 7232: ConditionalRequests
RFC 7233: Range Requests
RFC 7234: Caching
RFC 7235: Authentication

Roy Fielding est éditeur en chef de toutes ces spécifications

REST en pratique

Utilisation avancée de HTTP

Rappel : verbes HTTP

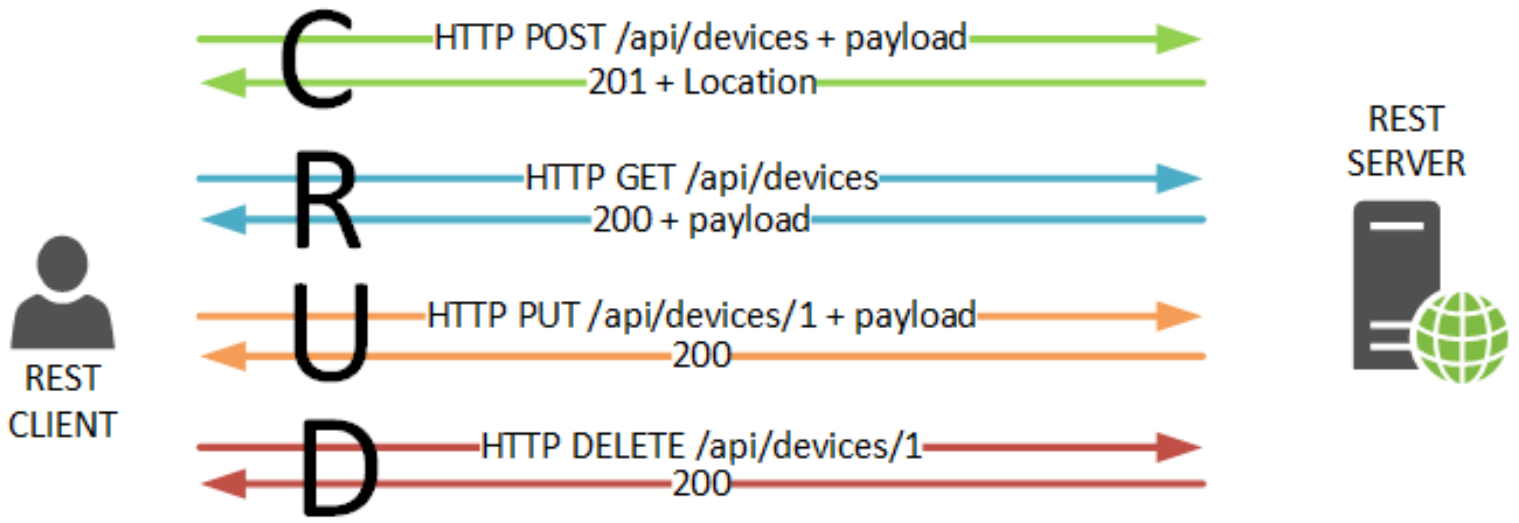
Sémantique des méthodes

Source : RFC 7231

Safe = sans effet de bord

	Safe	Idempotent	Cacheable
GET	X	X	X
POST			X
PUT		X	
DELETE		X	

Correspondance avec CRUD



REST en pratique

Utilisation avancée de HTTP

Remarques sur la sémantique des verbes HTTP

POST est officiellement définie comme cachable

- Sous réserve d'existence de certaines directives de cache
- Dans le cas d'une réponse 200 OK

  …mais très rarement implémenté

Contenu de réponse

- Seule GET a pour sémantique d'obtenir un contenu dans la réponse
- POST, PUT, DELETE visent à modifier le serveur ; un contenu dans la réponse n'est en général pas nécessaire

REST en pratique

Utilisation avancée de HTTP

Codes de statut / d'erreur

Bonnes pratiques pour les codes de statut (succès)

Renvoi d'un contenu :
200 OKou 206 Partial Content

Création d'une ressource :
201 Created+ lien vers la ressource dans le header Location

Modification d'une ressource :
200 OK+ représentation, si nécessaire
204 No Content, sinon

Suppression d'une ressource :
204 No Content

Ne pas hésiter à séparer les opérations des CU et renvoyer des codes de succès sans contenu.

REST en pratique

Utilisation avancée de HTTP

Codes de statut / d'erreur

Réactualisation d'une ressource :
304 Not Modified: le client peut réutiliser la dernière version renvoyée par le serveur (redirection vers le cache)
204 No Content: le client n'a pas besoin de modifier l'affichage, puisqu'aucun (nouveau) contenu ne provient du serveur

Erreurs d'authentification / d'autorisation
401 Unauthorized: erreur d'authentification
403 Forbidden: erreur d'autorisation (malgré une authentification possiblement correcte)

REST en pratique

Utilisation avancée de HTTP

"Although caching is an entirely OPTIONAL feature of HTTP, it can be assumed that reusing a cached response is desirable and that such reuse is the default behavior when no requirementor local configuration prevents it." – From RFC7234

Gestion du cache (RFC7234)

Spécifie le comportement des composants finaux et intermédiaires

- Rappel : la méthode HTTP utilisée doit être cachable

Notions de Freshness, d'âge, de Staleresponse(périmée)

- Headers : Age, Cache-Control, Expires, Warning

Fonctionnement : 2 techniques (requêtes conditionnelles)

- Last-Modified headers de requête : If-Modified-Since, If-Unmodified-Since, If-Range
- Etag(entitytag) headers de requête : If-Match, If-None-Match

Codes de réponse : 304 Not Modified, 412 PreconditionFailed(voir Validators)

REST en pratique

Utilisation avancée de HTTP

"When responses convey payload information, whether indicating a success or an error, the origin server often has different ways of representing that information;for example, in different formats, languages , or encoding" – from RFC7231

Négociation de contenus

- Rappel : client et serveur échangent des représentations de ressources

La négociation de contenus permet au serveur de renvoyer une représentation de ressource en fonction des préférences du client

- 2 façons pour le client d'exprimer ses préférences

  Headers HTTP

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en,en-US;q=0.8,fr;q=0.5,fr-FR;q=0.3
Accept-Encoding: gzip, deflate
```

  Extension dans l'URL

```
http://localhost/users/toto.html
http://localhost/users/toto.xml
http://localhost/users/toto.json
```

Remarque : dans ce cas, on spécifie uniquement le format de sérialisation

REST en pratique

Utilisation avancée de HTTP

Négociation de contenus

Envoi du media type(ex. MIME types) dans la réponse

Simple	`Content-Type:`	`application/json`
Suffixe de sous-type	`Content-Type:`	`application/ld+json`
Sous-type « Vendor »	`Content-Type:`	`application/vnd.ms-excel`
Sous-type « Personal or vanity »	`Content-Type:`	`audio/prs.sid`
Sous-type « unregistered »	`Content-Type:`	`application/x.foo`
Paramètres	`Content-Type:`	`text/html; charset=utf-8`

Syntaxe générale

type "/" subtype["+" suffix] *[";" parameter]

REST en pratique

Utilisation avancée de HTTP

Négociation de contenus

Plusieurs patterns de négociation dans la spec

- Proactive Negotiation: une seule transaction HTTP
  - Requête : headers Accept*
  - Réponse : headers Content*
- ReactiveNegotiation: deux transactions
  - .Réponse vide avec code : 300 Multiple Choices / 406 Not Acceptable -> listede choix
  - .Nouvelle requêteavec sélectionparmiles propositions

- Autres : Conditionalcontent, Active content, Transparent Content Negotiation…

- Concrètement :
  - Laisser le client exprimer ses préférences (ou pas) dans la requête initiale
  - Correctement implémenter la négociation proactive
  - Prévoir la négociation réactive en fallback(406 Not Acceptable)

REST en pratique

Utilisation avancée de HTTP

Déploiement indépendant des composants

- **Rappel : «scalabilité anarchique»**
- **Principe : pouvoir rajouter un composant de façon «transparente» pour l'application**
- **Fait appel à d'autres contraintes**
  - Composant terminal : serveur transactions sans état
  - Composant terminal : client interface uniforme
  - Composant intermédiaire : proxy, firewall, cache, loadbalancer… client-serveur, cache
    - Messages auto-descriptifs (interprétables par les intermédiaires)
    - Prise en compte des composants intermédiaires par HTTP
      Codes de retour (305 Use Proxy, 502 Bad Gateway)
      Directives de cache (no-store)
    - Certains composants se comportent à la fois comme un client et comme un serveur
      Séparation des préoccupations entre les 2 rôles

REST en pratique

Utilisation avancée de HTTP

Cas des cookies

Cookies de session

Contiennent uniquement un identifiant ; les données de session sont conservées côté serveur
violent le principe de Connexion sans état

Cookies contenant explicitement les «données de session»

Stateless, car les données sont bien envoyées dans la requête
Pas RESTfulcar le client n'est pas libre de choisir l'état de l'interaction(c'est le serveur qui impose la valeur des cookies)

Remarque : les cookies changent *a posteriori* la sémantique des requêtes précédemment effectuées
=>« cassent » le bouton back et certains mécanismes de cache

À proscrire en REST

REST en pratique

Authentification sans état

Rappel : la gestion des sessions

- Permet au serveur de « se rappeler » d'un utilisateur
  - Données de profil
  - Traces d'interaction
  - ...
- Techniquement
  - Le client n'envoie qu'un identifiant
    - Cookie
    - Token dans l'URL
    - Champ caché de formulaire
  - Le serveur maintient les données associées à cet identifiant : Ex (en Java) : Map

REST en pratique

Authentification sans état

Rappel : la gestion des sessions

- Avantages
  - Données protégées (côté serveur)
  - Bande passante réseau
- Inconvénients
  - Le client n'a pas accès à ses propres données
  - Scalabilité
    - Ressources consommées côté serveur
    - Goulot d'étranglement
- Comment faire pour
  - Transférer au client la responsabilité de gérer ses données
  - Satisfaire les contraintes fonctionnelles (sécurité, scalabilité...)

REST en pratique

Authentification sans état

Rappel : messages auto-descptifs

- Le client envoie le contexte de l'interaction avec chaque requête
- Y compris l'authentification

Problème :

- On ne peut pas demander à l'utilisateur de retaper ses identifiants à chaque requête
- Stocker son login et son mot de passe côté client

Solution

- L'utilisateur / l'agent s'identifie une fois
- Le serveur lui renvoie un token d'authentification
- Le client renvoie ce token à chaque requête ultérieure
- Le serveur vérifie qu'il s'agit bien de l'utilisateur en validant le token

REST en pratique

Authentification sans état

Stateful

Le token est une clé pour accéder aux variables de session

- Il est généré par le serveur, la seule restriction est qu'il soit unique
- Il doit être conservé côté serveur et côté client

Le serveur gère la session

- La session peut être invalidée par le serveur

Difficile de « partager » ce mécanisme entre plusieurs serveurs

- Il faut aussi déléguer toute la gestion des données de session

Stateless

Le token ne permet que la validation de l'identité de l'utilisateur

- Il est obtenu par chiffrement des informations d'authentification de l'utilisateur
- Il suffit d'avoir la clé pour déchiffrer le token et valider l'authentification

Le token a une durée de vie fixe

- Le serveur ne peut pas invalider la session

La gestion de l'authentification peut facilement être confiée à un tiers

- Exemples : SSO, CAS

REST en pratique

Authentification sans état

Stateless

Avantages

- Peu de mémoire consommée côté serveur
- Scalabilité

  Ajout d'un serveur : il suffit de lui partager la clé
- Externalisation
  - Single-Sign On (SSO)
  - Se loguer avec son compte Google ou Facebook...
  - Mécanismes d'authentification génériques (CAS, Oauth)

Inconvénients

- Protection de la clé
- Aucun contrôle sur
  - La révocation de session
  - La véracité / fraîcheur des données de session
- Implémentation plus lourde

  Surtout pour un seul serveur

REST en pratique

Authentification sans état

Authentification en HTTP (RFC 7235)

Requête à un serveur

En-tête Authorization

Indique une demande d'authentification du client au serveur

Contenu

- Type d'authentification (voir plus loin)
- Credentials(voir plus loin) :

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Requête à un proxy

En-tête **Proxy-Authorization**

Contenu : identique

REST en pratique

Authentification sans état

Authentification en HTTP (RFC 7235)

Réponse du serveur

Code d'erreur : **401 Unauthorized**

En-tête WWW-Authenticate

Permet à un serveur d'indiquer au client comment accéder à une ressource

Contenu

- Type d'authentification (scheme)
- Realm, charset… (facultatifs)

Réponse du proxy

Code d'erreur : **407 Proxy AuthenticationRequired**

En-tête : Proxy-Authenticate

Contenu : identique

REST en pratique

Authentification sans état

Types d'authentification en HTTP (Méthodes les plus classiques**)**

Scheme	Spec	Description
Basic	rRFC76i7l	Syntaxe = userid password ; encodé en base64
Bearer	rRFC675Ql	Token (voir plus loin)
Digest	[RFC7616l	Proche de Basic, mais le serveur indique au client comment formuler et **chiffrer** les credentials
OAuth	[RFC584Q, Section 3.5.1l	Utilise le protocole Oauth (voir plus loin)

REST en pratique

Authentification sans état

JSON Web Token (JWT)

Header

Objet JSON

Identifie l'algo de chiffrement de la signature

Payload

Objet JSON

Contient des « claims »

- Données d'authentification
- Ou autres

Signature

Avec une clé symétrique (HMAC)

Avec une clé asymétrique (RSA ou ECDSA)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikxpb
25lbCBMZWl2YSIsImlhdCI6MTUxNjIzOTAyMn0.
J48xFFrgRhAG-tlcvvKI98jPXw-
Wmuh10Nqv7h_Y618
```

REST en pratique

Authentification sans état

OAuth2

- RFC6749(core) et RFC8252(Native Apps)
- Protocole d'authentification et de délégation d'authentification (SSO)
  - 4 scénarios (granttypes)
  - Plusieurs types de cibles (webappscôté serveur, Single-Page-Applications, desktop, mobile…)
- Standardisé et largement utilisé
  - Google
  - Facebook
  - …
- Nombreuses implémentation disponibles
- Open source

REST en pratique

Hypermedia As The Engine of Application State - HATEOAS

Principe

"This architectural style lets you use hypermedia links in the response contents so that the client can dynamically navigate to the appropriate resource by traversing the hypermedia links. This is conceptually the same as a web user navigating through web pages by clicking the appropriate hyperlinks in order to achieve a final goal." - restfulapi.net

Exemple

Un client REST accède à une application REST à l'aide d'une simple URL fixe. Toutes les futures actions que le client peut entreprendre sont découvertes dans les représentations de la ressource retournée par le serveur. Les types de médias utilisés pour ces représentations, et les relations de liens qu'ils peuvent contenir, sont normalisés. Le client passe d'un état d'application à l'autre en sélectionnant des liens à l'intérieur d'une représentation ou en manipulant la représentation par d'autres moyens offerts par son type de média. De cette façon, l'interaction RESTful est pilotée par l'hypermédia, plutôt que par des informations hors bande.

REST en pratique

Hypermedia As The Engine of Application State - HATEOAS

Par exemple[1], cette requête GET récupère une ressource **account**, demandant des détails dans une représentation XML :

```
GET /accounts/12345/ HTTP/1.1
Host: bank.example.com
Accept: application/xml
...
```

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">100.00</balance>
    <link rel="deposit" href="/accounts/12345/deposit" />
    <link rel="withdraw" href="/accounts/12345/withdraw" />
    <link rel="transfer" href="https://bank.example.com/accounts/12345/transfer" />
    <link rel="close" href="https://bank.example.com/accounts/12345/close" />
</account>
```

La réponse contient les liens suivants : **deposit** (effectuer un dépôt), **withdraw** (effectuer un retrait), **transfer** (effectuer un transfert), ou **close** (clôturer le compte).

REST en pratique

Hypermedia As The Engine of Application State - HATEOAS

Lorsque les informations du compte sont récupérées ultérieurement, le compte est à découvert :

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">-25.00</balance>
    <link rel="deposit" href="/accounts/12345/deposit" />
</account>
```

Maintenant un seul lien est disponible : **deposit**. Dans son état actuel, les autres liens ne sont pas disponibles. D'où le *EOAS* de **HATEOAS**, pour *Engine of Application State* :
Les actions possibles varient en fonction de l'état de la ressource.

Un client n'a pas besoin de comprendre tous les types de médias et mécanismes de communication offerts par le serveur. La capacité de comprendre les nouveaux types de médias peut être acquise au moment de l'exécution grâce au "code à la demande" fourni au client par le serveur.

Web API's

REST fournit des principes pour

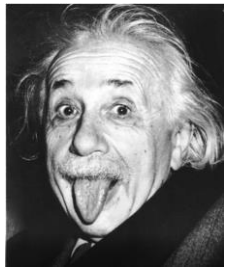
- o Alléger les serveurs
- o Permettre l'évolution des clients
- => Réduire le couplage entre serveur et client

« La théorie, c'est quand on sait tout et que rien ne fonctionne.

La pratique, c 'est quand tout fonctionne et que personne ne sait pourquoi.

Ici, nous avons réuni théorie et pratique :
Rien ne fonctionne... et personne ne sait pourquoi ! »

Albert Einstein



On peut donc réaliser des applications

- o En exposant des ressources sur un serveur
- o En documentant l'API du serveur Web
- o En réalisant le client séparément
  - * En JavaScript pour les navigateurs
  - * Sous forme d'applications desktop ou mobile
  - * Sur un autre serveur qui « consomme » les ressources du premier
  - * Sur des composants intermédiaires qui exposent / filtrent / transforment ces données

Web API's

D'ailleurs, a-t-on encore besoin de clients ?

Le modèle du Web s'est construit sur la mise à disposition *worldwide*de

- Documents (pour les humains)
- Données (pour les geeks et les machines)
- Services et Ressources (pour les machines)

Rappel de la notion d'affordance

- Permet de déduire comment on se sert de quelque chose, mais pas pourquoi
- Les objectifs des tâches réalisés par les clients sont rarement ceux prévus par les concepteurs des serveurs

Chacun son boulot

- Les serveurs **Web** exposent les ressources dont ils disposent et les **API** pour les utiliser
- Les clients utilisent ces API pour proposer aux utilisateurs des applications Web (ou pas) répondant au mieux à leurs besoins

Web API's

Vocabulaire

Un serveur expose une **Web API**

- Sous la forme d'un ensemble de ressources RESTful
- Avec un point d'entrée unique (URL de base)
- Avec sa documentation permettant
  - De comprendre comment elle s'utilise
  - De naviguer entre les ressources (HATEOAS)

Caractéristiques d'une Web API

Un client peut

- Consommer une Web API
- En agréger (**mashup**) plusieurs

Web API's

Accès aux Web API's

Sur le Web de documents, deux stratégies d'accès à l'information

- Navigation : liens dans les pages
- Requêtage: moteurs de recherche

Même problématique pour les Web APIs

- Navigation
  - Liens dans les APIs documentées connues
  - Liens rencontrés pendant l'utilisation d'une Web API
    - ✓ Headers HTTP : Link …
    - ✓ Redirections : 303 SeeOther…
- Requêtage: https://apislist.com/

Remarque

Toutes les APIs exposées sur le Web ne sont pas totalement RESTful

Web API's

Développer une Web API

Nombreux outils et frameworks orientés-ressources

En Java

JSR311 : Java API for RESTful Web Services (JAX-RS)

Définit principalement un ensemble d'annotations pour faciliter le développement en REST

- @Path, @PathParam, @QueryParam
- @GET,@POST,@DELETE,@PUT, @HEAD
- ©Produces,©Consumes

Implémentations : Apache CXF, Jersey, TomEE+...

Web API's

Outils d'aide au développement

Développer / débugger

- Côté serveur : utilisez le débugger de votre IDE
- Côté client : F12 (Outils de développement)

Documenter / tester une API

- https://swagger.io
- https://editor.swagger.io

Tester / scénariser

- https://www.soapui.org
- https://www.getpostman.com

Conclusion

REST, le modèle philosophique du Web ?

Un style architectural

- en accord avec la structure distribuée et l'échelle du Web
- pour la conception de services applicatifs sur le Web

Un ensemble de bonne pratiques

- Principes de conception
- Design patterns (RESTful, RESTlike, RESTafarian...)

Conclusion

Retour sur la notion d'application Web

Augmentation de la bande passante

- Toute l'information nécessaire au traitement de la requête doit « voyager » dedans

Nécessite des clients « intelligents »

- Mécanismes d'authentification plus lourds

Quid des données internes au serveur ?

Sémantique déclarative *vs.* opérationnelle

- Quid des CU qui s'écartent du modèle CRUD ?

Conclusion

REST a aussi des inconvénients

Manque d'un langage d'un standard de description d'interfaces

Fonctionnalités

- vérification automatique de la conformité
- génération automatique de code (squelettes, stubs)
- configuration automatique

De nombreuses propositions…

- WSDL 2.0 (2003)
- hRESTS et SA-REST (2007-2008)
- WADL (2009)
- HYDRA (2012)
- OpenAPI Specification (V1 : 2015, V3 : février 2020)

Mais aucune ne s'est encore complètement imposée.

Conclusion

Retour sur la notion d'application Web

Définition précédente

Application dont l'interface est visible dans un navigateur

Comment catégoriser les Web APIs ?

- (meilleure) utilisation des standards du Web
- Pas de client Web / pas de client du tout (Mais pas des applications)

Les Web APIs ne constituent pas des applications Web à part entière

- Modules d'applications Web
- À intégrer dans la démarche de conception d'une application (client)

Références

Grands principes

- https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/rest/
- https://www.crummy.com/writing/speaking/2008-QCon/act3.html
- https://martinfowler.com/articles/richardsonMaturityModel.html

Avancé

- https://fr.slideshare.net/AmruddhBhilvare/an-introduction-to-rest-api
- https: //fr.slideshare.net/stormpath/rest-jsonapis

URI / URL

- https: //restfulapi.net/resource-naming/

Références

Négociation de contenus / media types

- https://restfulapi.net/content-negotiation/
- https://akrabat.com/restful-apis-and-media-types/
- http://amundsen.com/media-types/
- https ://json-schema.org/latest/j son-schema-core.html

Authentification

o https: //medium.com/@kennch/stateful-and-stateless-authentication-ioaa2e2d4Q86 o https://jwt.io/ o https://oauth.net/

o https://aaronparecki.com/oauth-2-simplified/

o https://zestedesavoir.com/articles/1616/comDrendre-oauth-2-0-Dar-lexemDle/

Références

Méthodes HTTP

- https://ruben.verborgh.org/blog/2012/üq/27/the-obiect-resource-impedance-mismatch/

Web APIs

- https://rubenverborgh.github.io/WebFundamentals/web-apis/
- https://www.programmableweb.com/ o http: / /spec.openapis.org/oas/v3.o.3

JAX-RS

- https://jcp.org/en/jsr/detail?id=3ii
- https://en.wikipedia.org/wiki/Java API for RESTful Web Services
- http: //tomee.apache.org/tomcat-jaxrs.html

Talent HUB

Questions

Talent
HUB