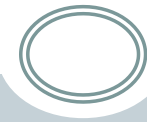


M1IFo3

Conception d'applications Web



REPRESENTATIONAL STATE TRANSFER (REST) ET WEB APIs

LIONEL MÉDINI
OCTOBRE-DÉCEMBRE 2020

Plan du cours

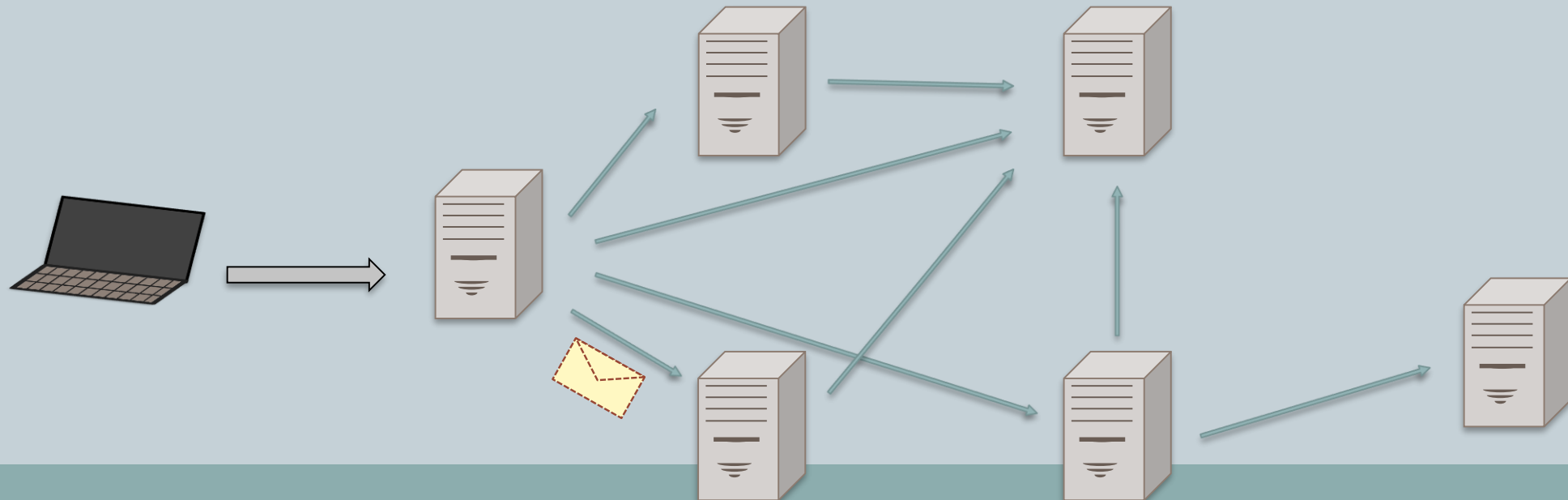


- **Introduction**
- REST en théorie
- REST en pratique
- Web APIs
- Conclusion

Avant : les Services Web



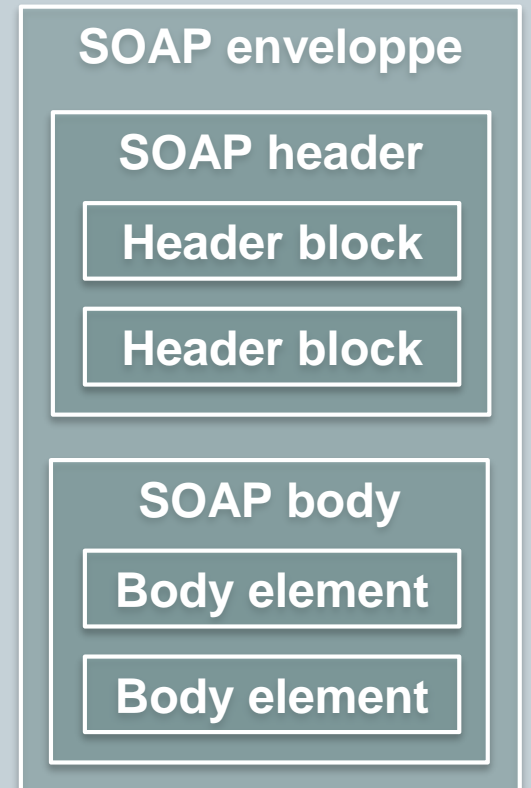
- Échange de **services** en Machine-to-Machine (M2M)
- Notion d'**Architectures Orientées-Services** (AOS)
- Notions d'« orchestration » et de « chorégraphie » de services
- Ensemble de technologies (SOAP, WSDL, UDDI, BPEL...)



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
 - ✦ Ex : SMTP, TCP, UDP, JMS...



Les services Web : au final



- Un mécanisme très puissant pour les applications Web complexes
 - Quid d'une application AJAX ?
 - 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

Plan du cours

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

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

Albert Einstein



Representational State Transfer (REST)



- Proposé en 2000 par Roy Fielding dans [sa thèse](#)
 1. Analyse des bonnes propriétés du Web
 2. Recherche des raisons de leur conception
 3. Généralisation sous forme d'un **style architectural** applicable au Web des machines
 4. Définition de **contraintes** pour le respect de ce style

Les bonnes propriétés du Web



- Rappel : le Web 1.0

HTML

→ Hypermédia

HTTP

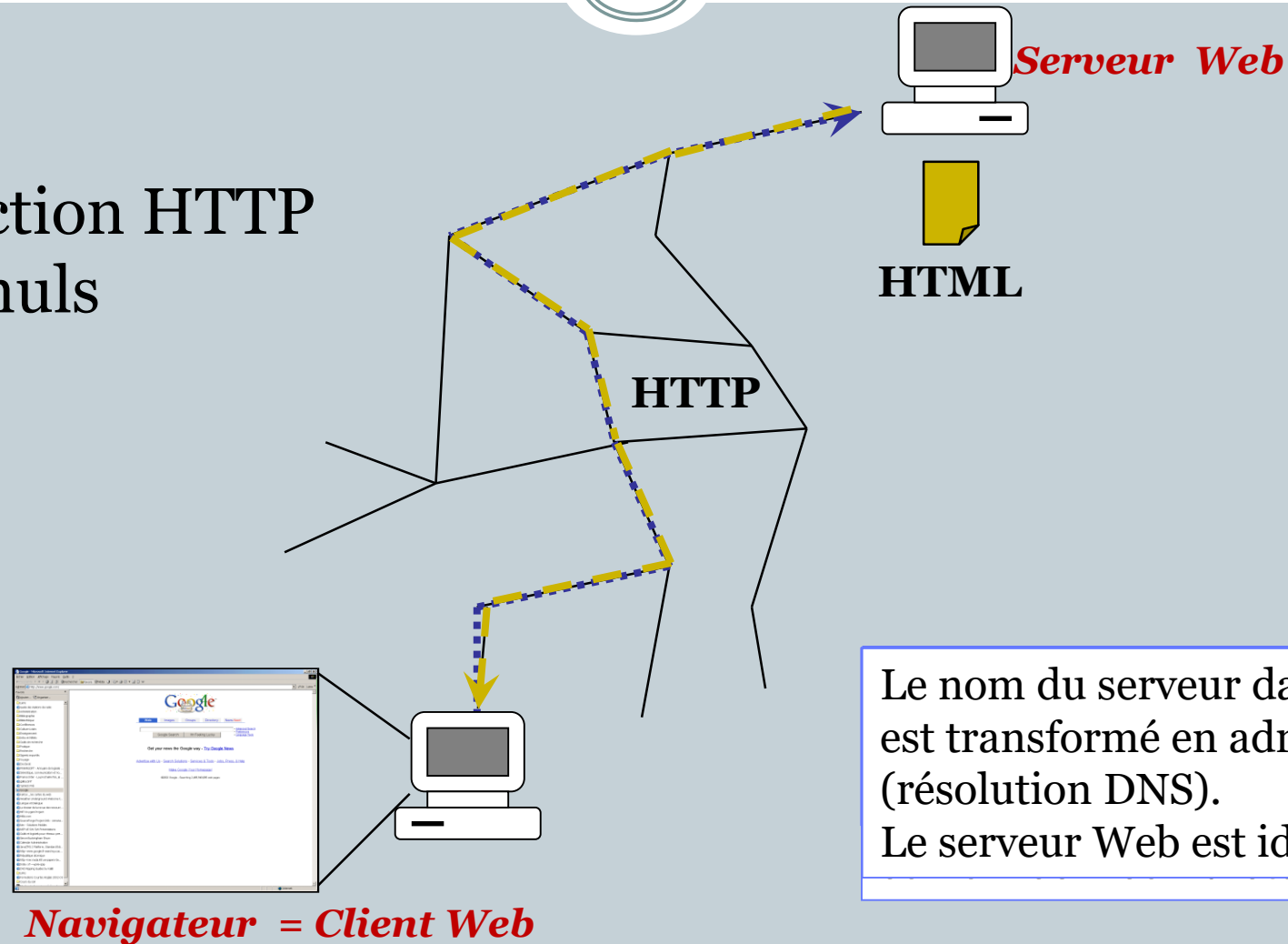
→ Verbes, codes de retour, media types, cache

URL

→ Ressource

Les bonnes propriétés du Web

- Rappel :
la transaction HTTP
pour les nuls

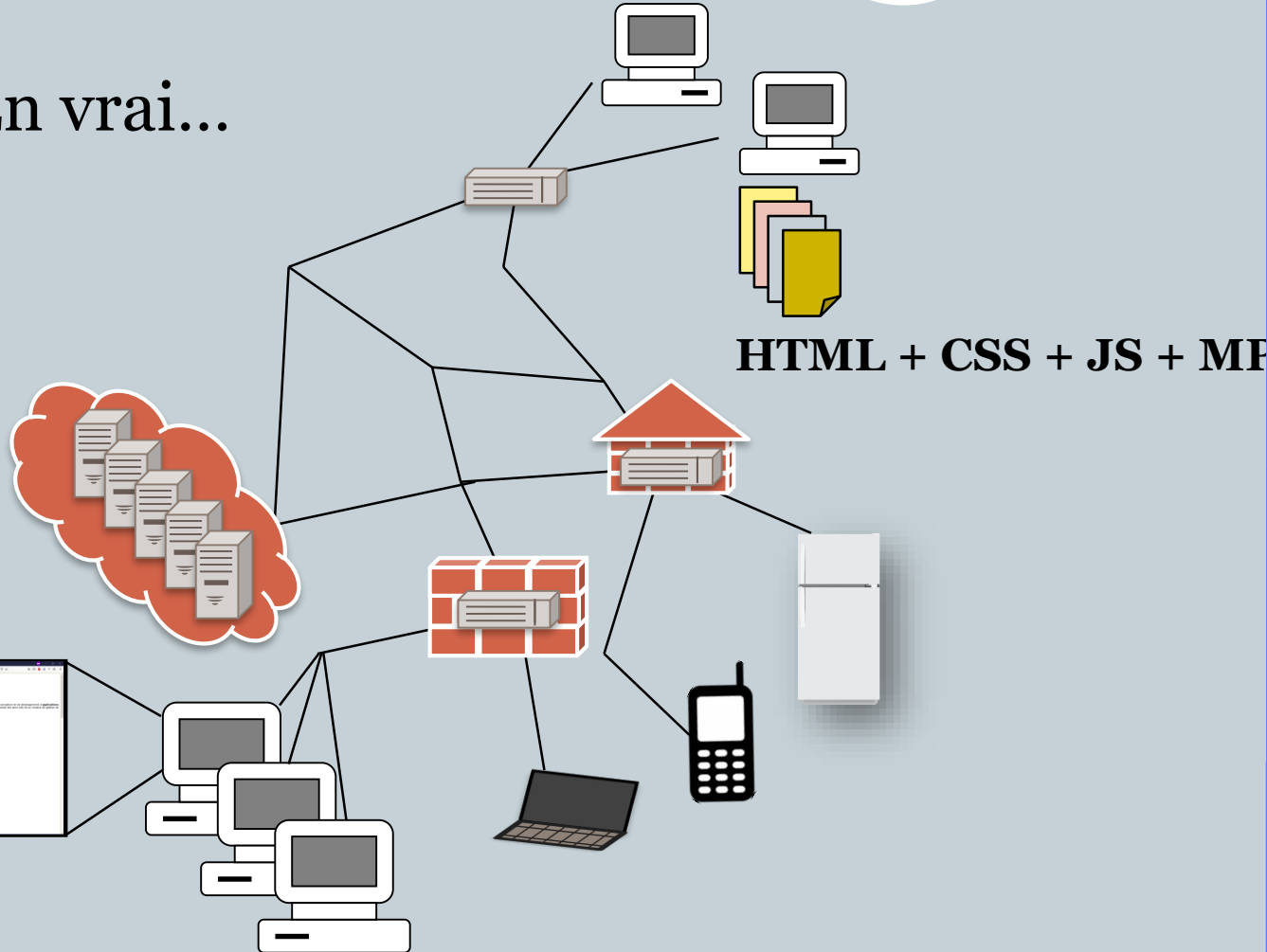


Le nom du serveur dans l'URL
est transformé en adresse IP
(résolution DNS).
Le serveur Web est identifié.

r)
e Web)

Les bonnes propriétés du

- En vrai...



Différents types de ressources :

- Augmentation de la complexité des applications

Plus de clients :

- Augmentation de la charge

Plus de serveurs :

- Réponse au besoin de scalabilité
- Spécialisation / load balancing

Des équipements intermédiaires (gateway, firewall, proxy, front server) :

- Prise en compte d'autres préoccupations (sécurité...)
- Gestion du cache

Des liens vers d'autres ressources potentiellement distantes (CDN, cloud...) :

- Navigation hypermédia
- Services externes

De nouveaux types de clients, avec des caractéristiques variables :

- connectivité réseau (latence),
- capacités de calcul / d'interaction,
- besoins applicatifs

Les bonnes propriétés du Web

- Abstraction

« WWW Application Domain Requirements :

- Low Entry-barrier
 - to enable sufficient adoption
- Extensibility
 - prepared for change
- Distributed Hypermedia
 - allows the [...] information to be stored at remote locations
 - designed for large-grain data transfer
- Internet-scale
 - anarchic scalability
 - independent deployment of software components »

(Roy Fielding, 2000)

REST, le style architectural du Web

- Généralisation

« **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, 2000)

REST, le style architectural du Web

- Contraintes (1/4)

- **Uniform interface**

- identification of resources
 - manipulation of resources through representations
 - self-descriptive messages
 - hypermedia as the engine of application state

(Roy Fielding, 2000)

- ➔ Conception orientée-ressources
- ➔ Changements d'états par des liens hypermédias

REST, le style architectural du Web

- Contraintes (2/4)

- **Client-Server**

- Separation of concerns

- **Stateless**

- each request must contain all of the information necessary to understand the request

(Roy Fielding, 2000)

→ Le serveur ne gère pas les états de l'interaction avec le client

REST, le style architectural du Web

- Contraintes (3/4)

- **Cache**

- a response [is] implicitly or explicitly labeled as cacheable or non-cacheable

- **Layered System**

- each component cannot "see" beyond the immediate layer with which they are interacting

(Roy Fielding, 2000)

→ Scalabilité

→ Intérêt des composants intermédiaires

REST, le style architectural du Web

- Contraintes (4/4)

- **Code-On-Demand** [optional]

- allows client functionality to be extended by downloading and executing code in the form of applets or scripts

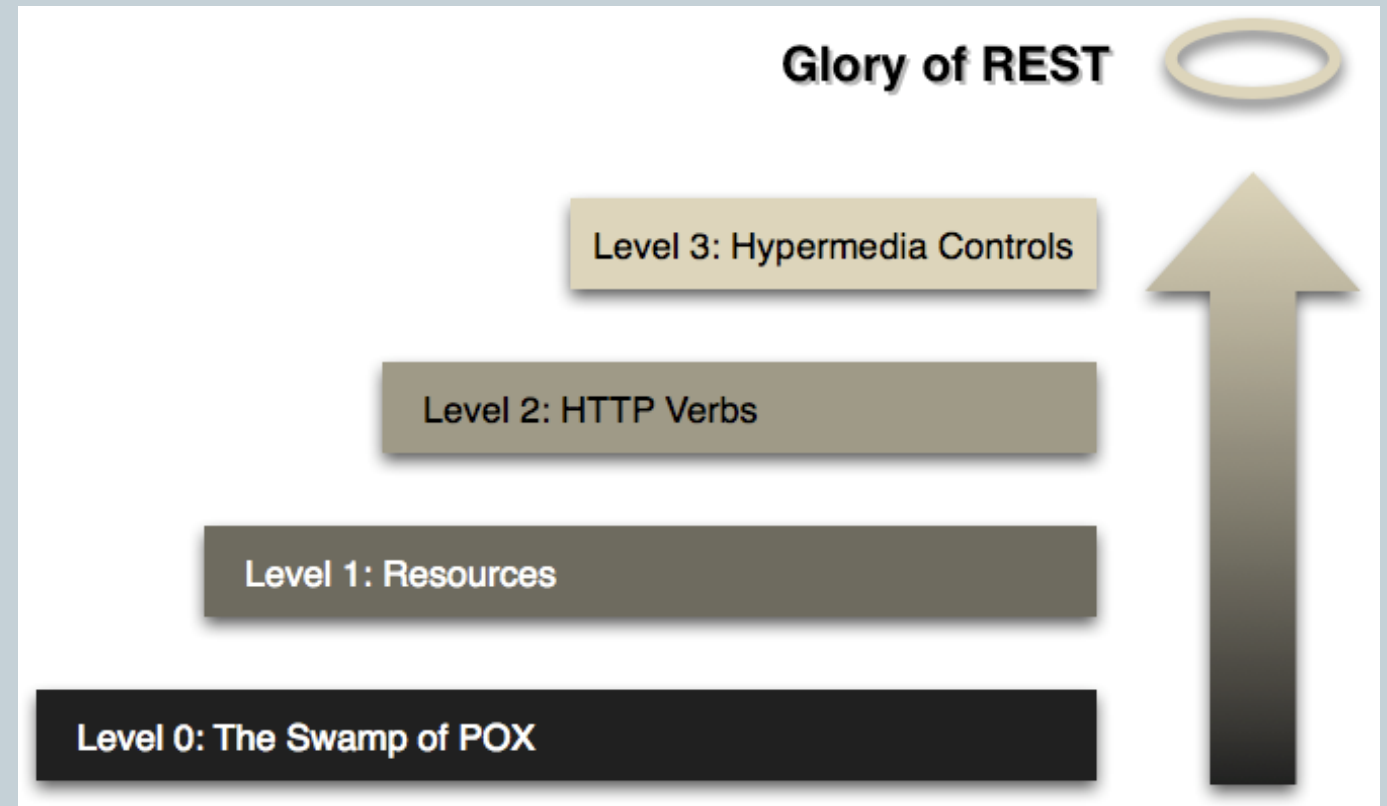
(Roy Fielding, 2000)

- ➔ Simplifie le travail du client
- ➔ Permet de faire évoluer l'interface

Modèle de maturité de Richardson



- ...
- Liens / navigation hypermédia
- Utilisation complète de HTTP
- Approche orientée-ressource
- Plain Old XML (SOAP)



Source : <http://martinfowler.com/articles/richardsonMaturityModel.html>
Plus de détails dans le [cours de PA Champin](#)

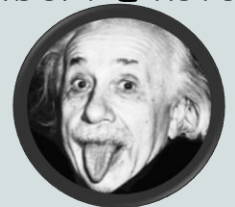
Plan du cours

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

« 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, concrètement

- Interface uniforme
- Client-Serveur
- Stateless
- Cache
- Système en couches

- Objectif de cette partie
 - « Traduire » les contraintes théoriques en « recettes » opérationnelles
 - Aborder les détails de mise en œuvre
- Contenu
 - Approche orientée-ressources
 - Transactions sans état
 - Forme des URL
 - Utilisation avancée de HTTP
 - Authentification sans état
 - Flux applicatif : HATEOAS
- Non abordé
 - Code-On-Demand

Approche orientée-ressource

- Client-Serveur
- Interface uniforme

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)

- 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 CUs 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

Approche orientée-ressource

- **Client-Serveur**
- **Interface uniforme**

- **Travailler en orienté-ressource**

- **Exercice**

- ✦ Quelles sont les ressources que vous allez exposer dans votre TP ?
 - ✦ Quels sont les cas d'utilisation qui y sont liés ?
 - ✦ Quel modèle arborescent en découle ?

Approche orientée-ressource

- **Client-Serveur**
- **Interface uniforme**

- **Travailler en orienté-ressource**

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

- ✦ Théoriquement, en REST, la forme des URL est libre

- Du moment qu'elles identifient une ressource de façon unique (d'où le nom)

- Exemples

- `https://monmoteurderecherche.com/search`

- `https://monreseausocial.com/user12345`

- Bonnes pratiques

- Faire apparaître le type et le nom de la ressource adressée

- Réfléter la hiérarchie de ressources

- `https://monreseausocial.com/users/user12345`

→ Exemples dans le TP ?

- 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
 - Cf. négociation de contenus

→ Exemple : dans le TP, préciser le contenu

- des requêtes
- des réponses

Transactions sans état

- **Stateless**

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

- **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(cf. scope des JavaBeans)

→ Exemple à partir du TP ?

Transactions sans état

- Stateless

- Travailler en « Stateless »

- 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":"français","pl":"ext-ff","lui":"français","cat":"web","sc":"Cgfw2mIu114S20"}
```

- ✦ Dans les headers HTTP

- Authorization

- Cookies (stateless, mais pas restful)

- ...

→ Exemple à partir du TP ?

- 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

- 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 slashes (« / »)
    - ✦ Pas de slash final

- Paramètres : filtres, tri, pagination

- Exemples

- ✦ `https://monserveur.com/api/v3/users/toto/friends/1`
    - ✦ `https://monserveur.com/api/v3/messages?author=toto&sort=+title,-index`
    - ✦ `https://monserveur.com/api/v3/messages?offset=100&limit=25`

- Bonnes pratiques

- 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://monserveur.com/users/login`

- `https://monserveur.com/users/toto/playlist/play`

- ✦ Pas toujours évident : que choisir ?

- `https://monserveur.com/users/logout`

- `https://monserveur.com/users/toto/logout`

# Utilisation avancée de HTTP

- Interface uniforme
- Client-Serveur
- Stateless
- Cache
- Système en couches

- Historique des spécifications IETF
  - Juin 1999
    - ✧ [RFC 2616](#) : contient tout HTTP 1.1
  - Juin 2014
    - ✧ [RFC 7230](#) : Message Syntax and Routing
    - ✧ [RFC 7231](#) : Semantics and Content
    - ✧ [RFC 7232](#) : Conditional Requests
    - ✧ [RFC 7233](#) : Range Requests
    - ✧ [RFC 7234](#) : Caching
    - ✧ [RFC 7235](#) : Authentication
    - ✧ Roy Fielding est éditeur en chef de toutes ces spécifications

# Utilisation avancée de HTTP

- Interface uniforme
- Cache

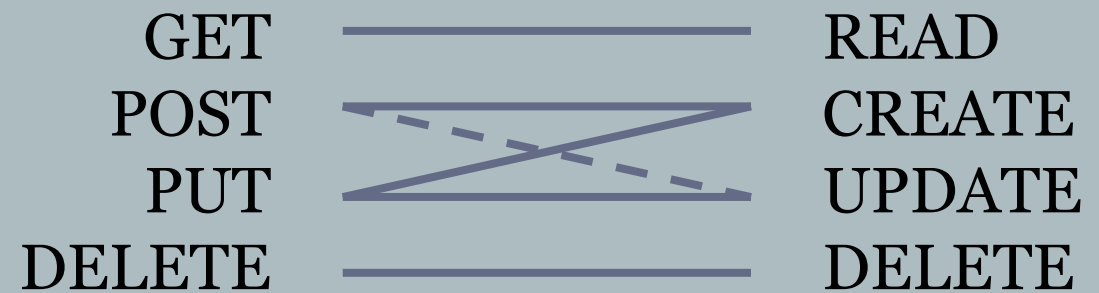
- Rappel : verbes HTTP

- 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





- 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
  - Bonne pratique
    - ✦ Préférer PUT à POST pour la mise à jour d'une ressource de type instance

- Codes de statut / d'erreur
  - Types de codes : voir [CM1](#)
  - Bonnes pratiques pour les codes de statut (succès)
    - ✦ Renvoi d'un contenu :
      - 200 OK ou 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. Pourquoi ?

- Codes de statut / d'erreur

- Quelques subtilités utiles pour renvoyer les bons codes dans les TP

- ✦ 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

- ➔ Exemples pour le TP ?

- ✦ Erreurs d'authentification / d'autorisation

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

- ➔ Impact sur le TP ?

- Gestion du cache ([RFC7234](#))

"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 requirement or local configuration prevents it." - [RFC7234](#)

- Spécifie le comportement des composants finaux et intermédiaires
  - ✦ Rappel : la méthode HTTP utilisée doit être cachable
- Notions de [Freshness](#), d'[âge](#), de [Stale response](#) (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](#) (entity tag) → headers de requête : [If-Match](#), [If-None-Match](#)
- Codes de réponse : [304 Not Modified](#), [412 Precondition Failed](#) (voir [Validators](#))

- Négociation de contenus

- Rappel : client et serveur échangent des représentations de ressources

"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 encodings." - [RFC7231](#)

- ✦ La négociation de contenus permet au serveur de renvoyer une représentation de ressource en fonction des préférences du client
- ➔ Quel est le design pattern incontournable pour implémenter la négociation de contenus ?

- Négociation de contenus

- 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

- Négociation de contenus

- Envoi du [media type](#) (ex. MIME types) dans la réponse

- ✦ Simple
    - ✦ Suffixe de sous-type
    - ✦ Sous-type « Vendor »
    - ✦ Sous-type « Personal or vanity »
    - ✦ Sous-type « unregistered »
    - ✦ Paramètres

```
Content-Type: application/json
```

```
Content-Type: application/ld+json
```

```
Content-Type: application/vnd.ms-excel
```

```
Content-Type: audio/prs.sid
```

```
Content-Type: application/x.foo
```

```
Content-Type: text/html; charset=utf-8
```

```
Content-Type: application/vnd.oai.openapi; version=2.0
```

```
Content-Type: application/x.foo+json; schema="https://example.com/my-hyper-schema#"
```

- Syntaxe générale

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

- Négociation de contenus

- Plusieurs patterns de négociation dans la spec

- ✦ [Proactive Negotiation](#) : une seule transaction HTTP

- Requête : headers Accept\*

- Réponse : headers Content\*

- ✦ [Reactive Negotiation](#) : deux transactions

- 1. Réponse vide avec code : 300 Multiple Choices / 406 Not Acceptable -> liste de choix

- 2. Nouvelle requête avec sélection parmi les propositions

- ✦ Autres : Conditional content, 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)



- **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, load balancer... ← 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

- 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 RESTful car 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

- **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

- **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...)

# Authentification sans état

• **Stateless**

- **Rappel : messages auto-descriptifs**
  - 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
    - ✦ ...en particulier en AJAX
  - 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

Bravo !  
Mais ce n'est pas déjà  
exactement ce qu'on  
fait quand on gère  
des sessions ?...

# Authentification sans état

- **Stateless**

## 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

# 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

# Authentification sans état

• **Stateless**

- 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)

```
WWW-Authenticate: Basic realm="admin", charset="UTF-8"
```

- Réponse du proxy

- ✦ Code d'erreur : 407 Proxy Authentication Required
    - ✦ En-tête : Proxy-Authenticate
    - ✦ Contenu : identique



# Authentification sans état

- **Stateless**

- 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

# Authentication sans état

- **Stateless**

- Types d'authentification en HTTP
  - Méthodes les plus classiques

| Scheme | Spec                                       | Description                                                                                            |
|--------|--------------------------------------------|--------------------------------------------------------------------------------------------------------|
| Basic  | [ <a href="#">RFC7617</a> ]                | Syntaxe = userId ":" password ; encodé en base64                                                       |
| Bearer | [ <a href="#">RFC6750</a> ]                | Token (voir plus loin)                                                                                 |
| Digest | [ <a href="#">RFC7616</a> ]                | Proche de Basic, mais le serveur indique au client comment formuler et <b>chiffrer</b> les credentials |
| OAuth  | [ <a href="#">RFC5849, Section 3.5.1</a> ] | Utilise le protocole Oauth (voir plus loin)                                                            |

# Authentification sans état

- **Stateless**

- JSON Web Token (JWT)

- Techniquement

- ✦ 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)

```
{
 "typ": "JWT",
 "alg": "HS256"
}
```

```
{
 "iss": "https://monserveur.com",
 "sub": "1234567890",
 "name": "Lionel Médini",
 "iat": 1516239022,
 "exp": 1516242622
}
```

➔ Test : <https://jwt.io/>

# Authentification sans état

• **Stateless**

- JSON Web Token (JWT)

- Intérêt

- ✦ Le client peut « lire » les données envoyées dans le token
      - Ou pas → JWE (encryption)
    - ✦ Mais il ne peut pas les modifier

- Faille (corrigée)

- ✦ En modifiant le header, il peut changer le type de chiffrement
    - ✦ Voir : <https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/>

- Intérêt de l'encodage en base64

- ✦ Permet de passer le token dans une URL

- Inconvénient de base64

- ✦ Inutile dans certains cas → Spécification JSON Web Signature (JWS)
      - [RFC 7797](#) (updates JWT)
      - Permet de spécifier si on utilise ou non base64 par un claim supplémentaire ("b64": false)

- OAuth2

- [RFC6749](#) (core) et [RFC8252](#) (Native Apps)
- Protocole d'authentification et de délégation d'authentification (SSO)
  - ✦ 4 scénarios (grant types)
  - ✦ Plusieurs types de cibles (webapps côté serveur, Single-Page-Applications, desktop, mobile...)
- Standardisé et largement utilisé
  - ✦ Google
  - ✦ Facebook
  - ✦ ...
- Nombreuses implémentations disponibles
- Open source

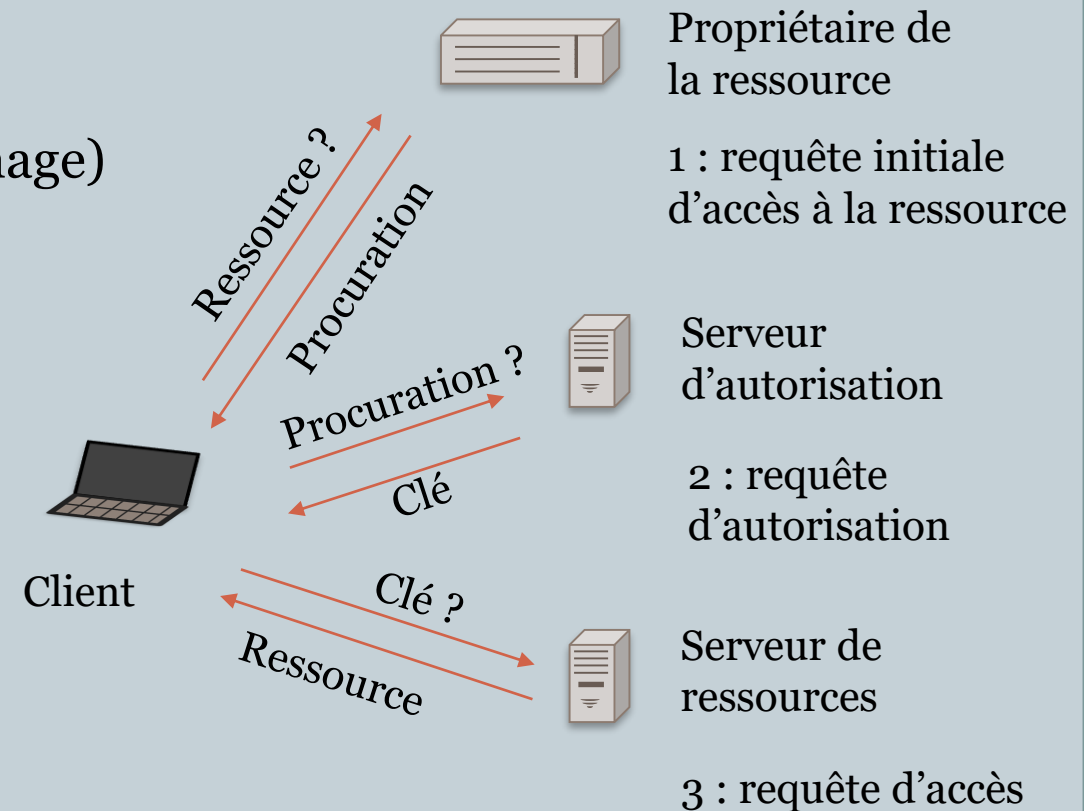
# Authentification sans état

• **Stateless**

## • OAuth2

### ○ Schéma général

- ✦ Exemple de demande d'accès à une ressource (image)
- ✦ Plusieurs acteurs peuvent être les mêmes
  - Propriétaire et serveur de ressource
  - Serveurs d'autorisation et de ressource
  - Client et propriétaire
- ✦ La clé peut être un token JWT




- [OAuth2](#)

- Aperçu technique

- ✦ S'appuie sur d'autres protocoles
      - Authentication : OpenID, SAML
      - SSO : SAML
    - ✦ Couche de sécurité : TLS (contre SSL pour Oauth 1)
    - ✦ Types de tokens
      - Oauth 2 token
      - JWT
    - ✦ Autres fonctionnalités
      - Révocation de session
      - Refresh token

# Hypermedia As The Engine Of Application State (HATEOAS)



- **Interface uniforme**
- **Stateless**
- **Client-serveur**

## • 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](http://restfulapi.net)

- « Engine Of Application State » = gestion du flux applicatif
  - ✦ Enchaînement des requêtes
  - ✦ Changements d'états du modèle
- ➔ Pas de gestion du flux applicatif par le serveur
  - ➔ Le serveur se contente de « proposer » des interactions en fournissant des liens
- ➔ Le client choisit « librement » quels liens suivre pour exécuter l'application
  - ➔ Principe « follow your nose »

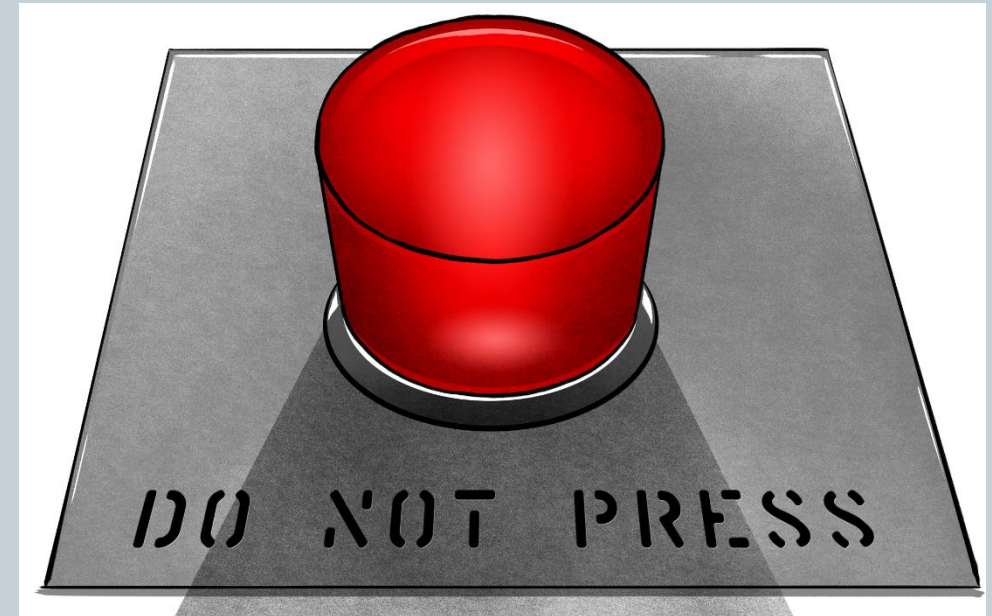


# Hypermedia As The Engine Of Application State (HATEOAS)

- Interface uniforme
- Stateless
- Client-serveur

- Notion d'affordance

- Le client doit pouvoir « comprendre » la sémantique des liens
  - ✦ Destination (ressource)
  - ✦ Mode d'utilisation (méthode, paramètres)
  - ✦ Effet attendu (représentation, changement d'état)



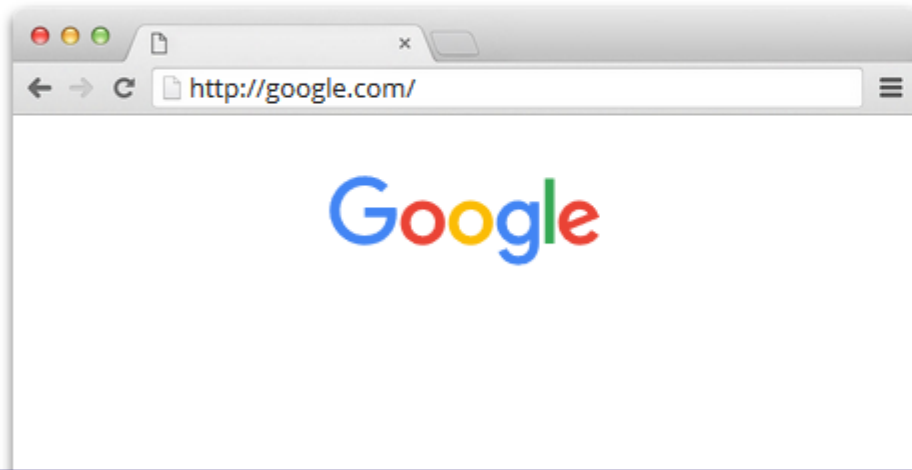
Source : <https://i-love-png.com/>

# Hypermedia As The Engine Of Application State (HATEOAS)

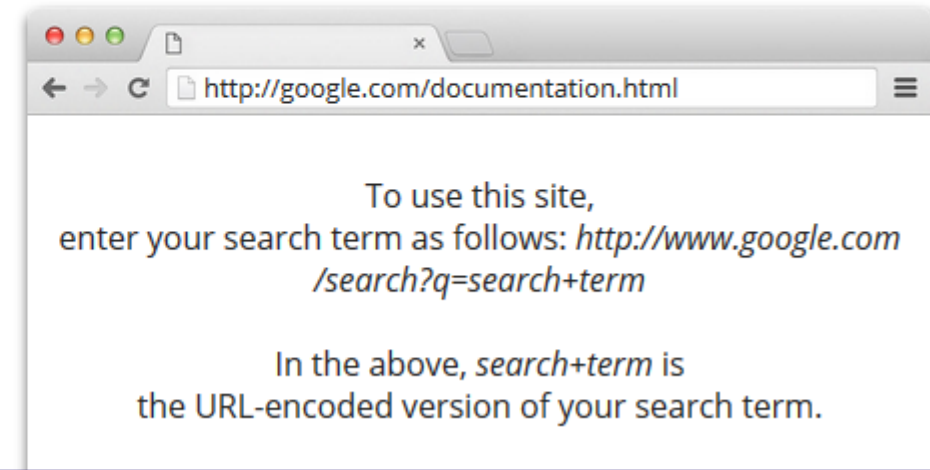
- **Interface uniforme**
- **Stateless**
- **Client-serveur**

- **Contrôles hypermédias**

To understand hypermedia's importance, imagine using any website without it.




You'd need to read the documentation, and hard-code it in your implementation.



Source : <https://rubenverborgh.github.io/WebFundamentals/web-apis/>

# Hypermedia As The Engine Of Application State (HATEOAS)



- **Interface uniforme**
- **Stateless**
- **Client-serveur**

- **Fonctionnement**

1. Le serveur répond aux requêtes en renvoyant des contrôles hypermédias
  - ✦ Contenant
    - Des liens (URI) vers les ressources
    - Éventuellement, une documentation des actions sur ces ressources
      - Si l'utilisation n'est pas évidente (CRUD)

# Hypermedia As The Engine Of Application State (HATEOAS)

- Interface uniforme
- Stateless
- Client-serveur

## • Fonctionnement

### 1. Le serveur répond aux requêtes en renvoyant des contrôles hypermédias

#### ✦ En utilisant

#### ○ Des headers HTTP

Location: `https://monserveur.com/users/toto`

Link: `<https://monserveur.com/users/titi>; rel="previous"; title="previous user"`

#### ○ Le corps de la réponse

```
{ "userId": "toto",
 "login": "toto",
 "links": [
 { "href": "https://monserveur.com/users/toto/messages",
 "rel": "messages",
 "type": "GET" }] }
```

# Hypermedia As The Engine Of Application State (HATEOAS)

- Interface uniforme
- Stateless
- Client-serveur

- **Fonctionnement**

2. Le client parcourt les liens grâce aux contrôles hypermédias...  
Oui, mais comment ?

- ✦ En présentant l'information à l'utilisateur et en le laissant choisir
- ✦ En « hard-codant » la logique applicative côté client
  - Mais à quoi sert la description dans ce cas ?
- ✦ En utilisant la description pour déduire automatiquement les actions à effectuer
  - La description doit être sérialisée sous forme d'un media type
  - Ce media type doit lui-même être conforme à un schéma
    - Simple : [JSON Schema](#), [XML Schema](#)...
    - Hypermédia : [HAL](#), [Collection+JSON](#), [JSON-API](#), [Siren](#), etc.
  - Voir : <https://akrabat.com/restful-apis-and-media-types/>

# Plan du cours

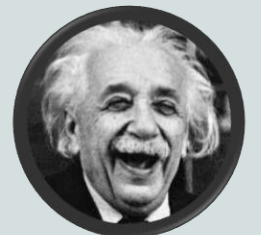
- Introduction
- REST en théorie
- REST en pratique
- **Web APIs**
- Conclusion

*« 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



# Web APIs



- REST fournit des principes pour
  - Alléger les serveurs
  - Permettre l'évolution des clients
  - ➔ Réduire le couplage entre serveur et client
- On peut donc réaliser des applications
  - En exposant des ressources sur un serveur
  - En documentant l'API du serveur Web
  - 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 APIs



- 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 ? 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 APIs



- 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

# Accès aux Web APIs



- Sur le Web de documents, deux stratégies d'accès à l'information
  - Navigation : liens dans les pages
  - Requête : 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 See Other` ...
  - Requête : <https://www.programmableweb.com/>
- Remarque
  - Toutes les APIs exposées sur le Web ne sont pas totalement RESTful

# 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
        - ...
      - ...Que vous n'utiliserez pas en TP
    - ✦ Implémentations
      - [Apache CXF](#), [Jersey](#), [TomEE+](#)...

# Outils d'aide au développement



- Développer / débbugger
  - Côté serveur : utilisez le débbugger de votre IDE
  - Côté client : F12
- Documenter / tester une API
  - <https://swagger.io/>
  - <https://editor.swagger.io/>
- Tester / scénariser
  - <https://www.soapui.org/>
  - <https://www.getpostman.com/>

# Plan du cours



- Introduction
- REST en théorie
- REST en pratique
- Web APIs
- 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 bonnes pratiques
  - Principes de conception
  - Design patterns
- Une religion
  - RESTful
  - RESTlike
  - RESTafarian...

**« Sur les chemins de l'illumination,  
nous sommes tous des canards en plastique. »**

(Pierre-Antoine Champin, 2019)

# REST a aussi des inconvénients



- 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 ?

# 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.



# 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
    - 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/AniruddhBhilvare/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/json-schema-core.html>
- **Authentification**
  - <https://medium.com/@kennch/stateful-and-stateless-authentication-10aa3e3d4986>
  - <https://jwt.io/>
  - <https://oauth.net/>
  - <https://aaronparecki.com/oauth-2-simplified/>
  - <https://zestedesavoir.com/articles/1616/comprendre-oauth-2-0-par-lexemple/>

# Références



- **Méthodes HTTP**

- <https://ruben.verborgh.org/blog/2012/09/27/the-object-resource-impedance-mismatch/>

- **Web APIs**

- <https://rubenverborgh.github.io/WebFundamentals/web-apis/>
- <https://www.programmableweb.com/>
- <http://spec.openapis.org/oas/v3.0.3>

- **JAX-RS**

- <https://jcp.org/en/jsr/detail?id=311>
- [https://en.wikipedia.org/wiki/Java\\_API\\_for\\_RESTful\\_Web\\_Services](https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services)
- <http://tomee.apache.org/tomcat-jaxrs.html>