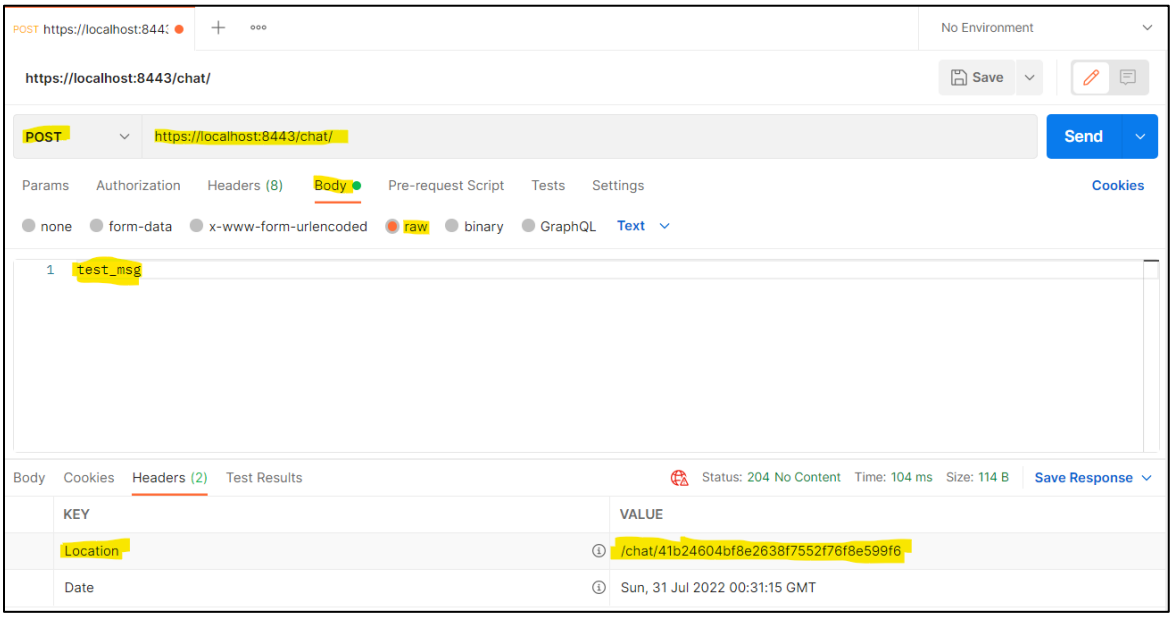
Master 2 **Protocoles des services Internet**

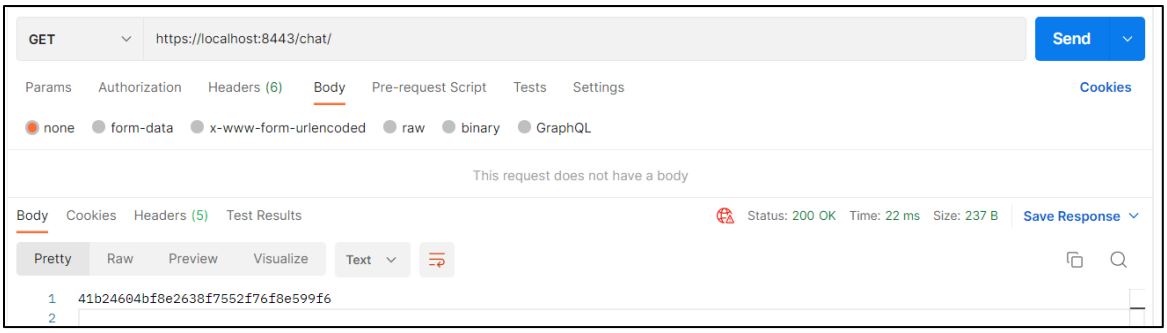
TP2 **Un client REST**

- Une interface REST :
  - **Une requête GET à l'URL /chat/** retourne la liste des identificateurs des messages du chat, en ordre chronologique;
  - **Une requête POST à l'URL /chat/** crée un nouveau message, et retourne son identificateur;
  - **Une requête GET à l'URL /chat/id** retourne la valeur d'un message de chat; la date du message est retournée dans l'entête **Last-Modified**;
  - **Une requête DELETE à la même URL** supprime un message.

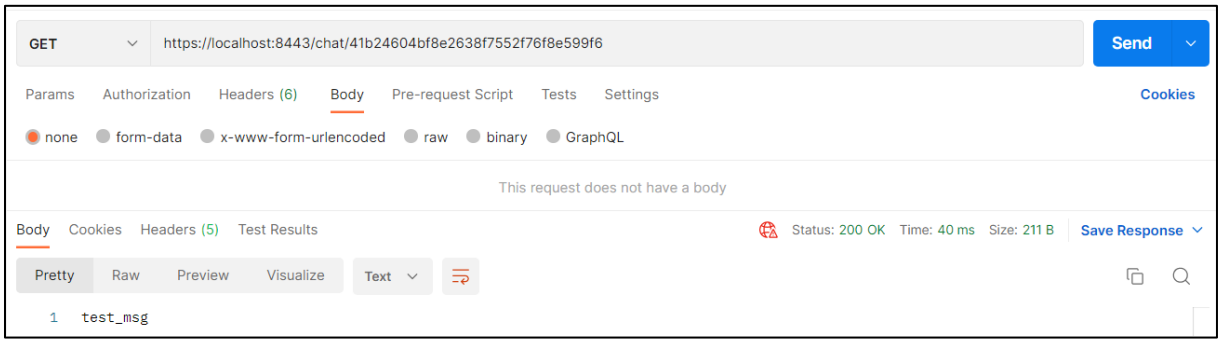
- **Une requête POST à l'URL /chat/** crée un nouveau message, et retourne son identificateur;

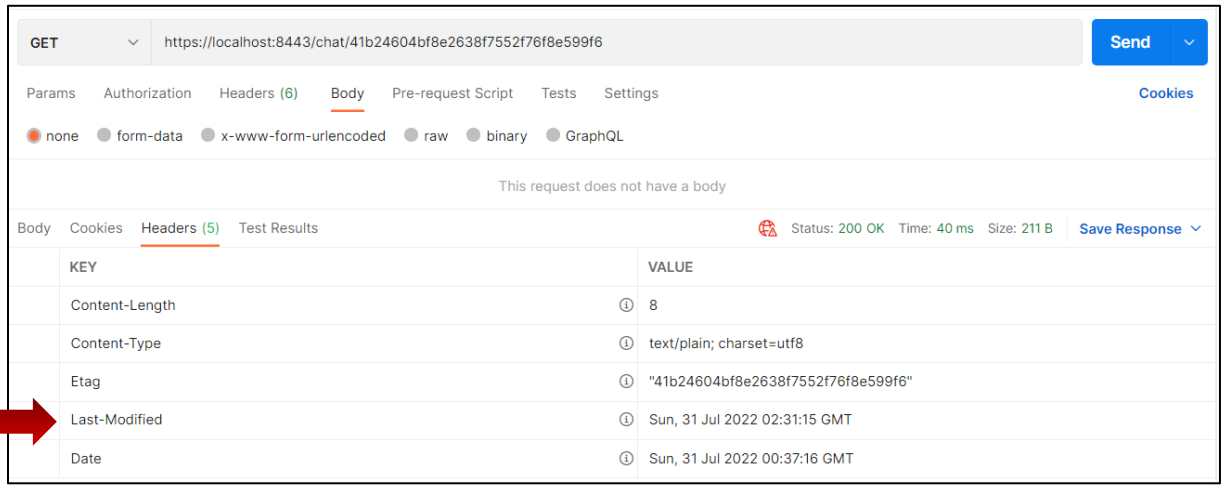


- **Une requête GET à l'URL /chat/** retourne la liste des identificateurs des messages du chat, en ordre chronologique;

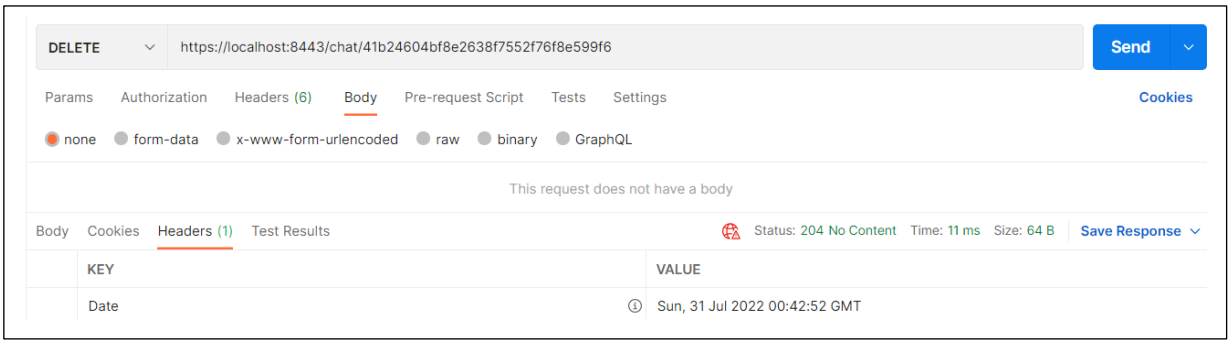


Une requête GET à l'URL /chat/id retourne la valeur d'un message de chat; la date du message est retournée dans l'entête **Last-Modified**;





Une requête DELETE à la même URL supprime un message.



Exercice 1.

Expliquez pourquoi cette interface est sans état de session (du côté serveur).

Le sans état : L'URI (*Uniform Resource Identifier*), qui identifie une ressource, et ses éventuels paramètres sont suffisants pour afficher une ressource ou effectuer une action sur une ressource. Il ne doit pas y avoir de notion de session, et les cookies ne doivent pas servir non plus à stocker un état.

REST est-elle une architecture *stateless* (sans état) ou *statful* (avec état) ?
La réponse est « *stateless* ». Il n'y a pas de notion de session (au sens large du terme) : le serveur ne se souvient pas des enchainements des requêtes, ne fait pas varier les réponses selon un enchainement particulier et ne stocke pas d'informations autres que les actions demandées sur les ressources (création, modification, suppression…).

Source : *Développer des services REST en Java : Échanger des données au format JSON / Sobrero Aurélie*

L'approche REST : il n'y a pas d'état de session du côté serveur — l'état de session doit être maintenu côté client, ou alors stocké dans des structures de données indiquées explicitement par le client.

Ce principe (pas d'état de session) a des conséquences importantes, il permet de facilement distribuer l'application sur plusieurs serveurs, de paralléliser les actions, ou encore de survivre au reboot d'un serveur; c'est un vieux principe de réseau, utilisé notamment dans le protocole NFS (publié en 1984).

Source : *Poly du cours / Juliusz Chroboczek*

Exercice 4(b).
Combien de requêtes HTTP votre programme fait-il ?

Au plus 51 requêtes http [1 requête à **/chat/** et puis au plus 50 requêtes à **/chat/id**].

Exercice 4(c).
Serait-il possible de les exécuter en parallèle ?

Oui, il est possible s'exécuter en parallèle les 50 requêtes à **/chat/id** car l'interface est sans état de session du coté serveur, c-à-d : *(cf. exercice 1)*

L'**id** qui identifie une ressource est suffisant pour afficher une ressource. Le serveur ne se souvient pas des enchainements des requêtes et ne fait pas varier les réponses selon un enchainement particulier.

Ce principe (pas d'état de session), permet de paralléliser les actions.

Exercice 4(d).
Quel est le nombre de RTT minimal ?

RTT : Round-trip Time ; la durée du trajet aller-retour

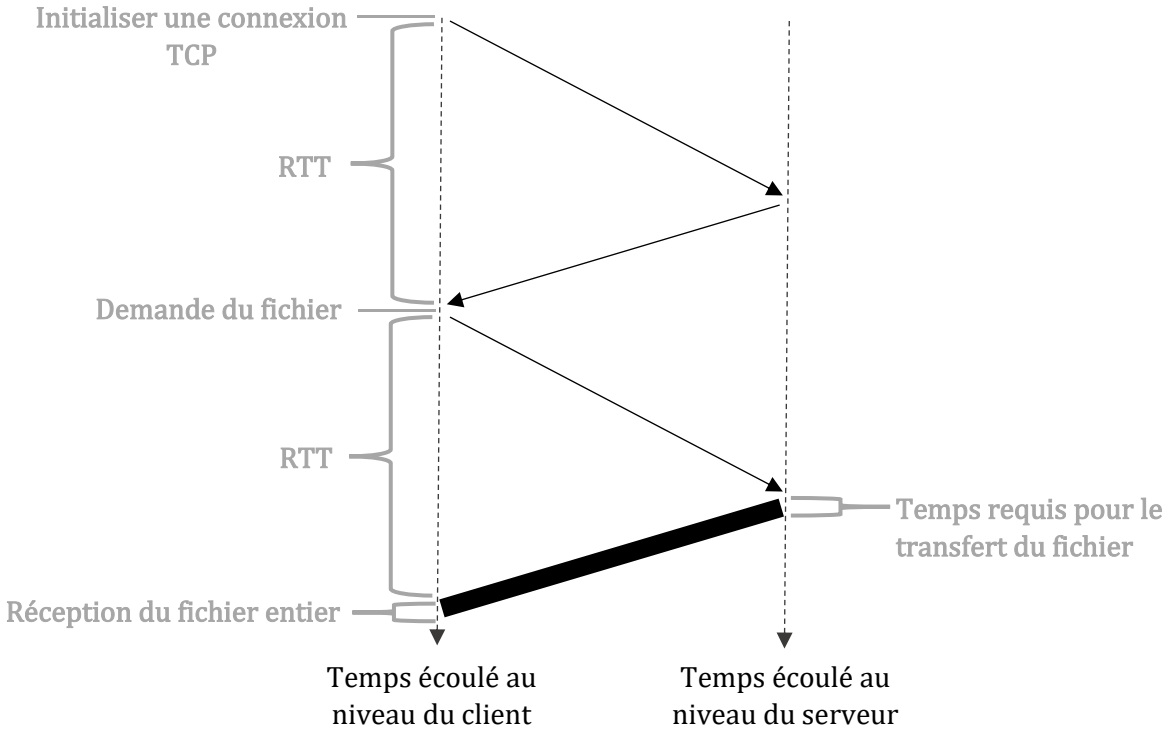
Essayons brièvement d'estimer le temps qu'il faut à un client pour obtenir le fichier qu'il sollicite. Dans ce but, nous définissons la durée du trajet aller-retour (RTT, Round-trip time), comme le temps nécessaire à un petit paquet pour faire un aller-retour entre le client et le serveur.

Ce RTT est la somme du temps de propagation, du temps d'attente au niveau des divers routeurs et commutateurs intermédiaires et du temps de traitement des paquets.

Considérez maintenant ce qui se produit lorsque l'utilisateur clique sur un hyperlien. Ceci conduit le navigateur à établir une connexion TCP avec le serveur Web, opération qui implique un « échange de présentation en trois

étapes », au cours duquel le client envoie un petit segment TCP au serveur, attend un accusé de réception de sa part (sous la forme d'un autre segment TCP) et l'informe finalement de la bonne réception de la réponse.

Le RTT se limite aux deux premières phases de la procédure d'échange de présentations. A l'issue de ces deux phases, le client envoie le message de demande http associé à la troisième étape (l'accusé de réception) sur la connexion TCP. Une fois ce message arrivé au serveur, ce dernier répond au moyen du fichier HTML sollicité. Cette procédure de demande/réponse implique un nouveau RTT, ce qui porte le temps d'attente total à environ deux RTT plus le temps de transmission du fichier HTML depuis le serveur.



Estimation du temps d'attente associé à la requête d'un fichier HTML

Connexions persistantes

Les connexions non-persistantes présentent certains inconvénients. Tous d'abord, une nouvelle connexion doit être établie et maintenue pour chaque objet sollicité. La livraison de chaque objet implique un temps d'attente de deux RTT : un premier lors de l'établissement de la connexion TCP et un second pour la requête et la réception de l'objet.

Avec des connexions persistantes, la connexion TCP est maintenue par le serveur après l'envoi de la réponse. Par conséquent, tous les échanges entre un client et un serveur peuvent emprunter la même connexion.

Il existe deux types de connexions persistantes : les connexions dites **avec** ou **sans pipelinage.** En l'absence de pipelinage, le client émet une nouvelle requête uniquement après la réception de la réponse à sa requête

précédente. Dans ce cas il subit un RTT pour la requête et la réception de chaque objet contenu dans la page qu'il sollicite. Bien que ceci représente déjà un progrès sensible par rapport aux deux RTT par objet associés aux connexions non-persistantes, la méthode du pipelinage permet encore de réduire le RTT.

Le mode par défaut de la version 1.1 de http utilise des connexions persistantes avec pipelinage. Dans cette configuration, le client http émet une requête dès qu'il rencontre une référence à un objet. Il est ainsi en mesure d'émettre des requêtes successives dans le cas ou la page sollicitée compterait plusieurs objets, c'est-à-dire qu'il peut émettre une nouvelle requête avant même d'avoir reçu la réponse à sa requête précédente.

Le serveur, lui, envoie les objets au rythme auquel lui arrivent les requetes. Avec le pipelinage, il est donc possible de ne générer qu'un seul RTT pour l'ensemble des objets référencés (au lieu d'un RTT par objet).

Source : *Analyse structurée des réseaux : des applications de l'internet aux infrastructures de télécommunication / James Kurose et Keith Ross*

HTTP/1.1

HTTP/1.1, normalisé en 1997, résoud la plupart des problèmes de HTTP/1.0 tout en restant compatible avec ce dernier. Tout d'abord, HTTP/1.0 utilise une connexion par « entité » transférée, ce qui utilise TCP d'une façon inefficace; HTTP/1.1 permet de réutiliser la même connexion pour plusieurs entités.

HTTP/2

HTTP/2 conserve la sémantique de HTTP/1.1 (toute requête HTTP/2 peut être convertie en HTTP/1.1 sans perte d'informations), mais change la syntaxe : les entêtes sont envoyés en un format binaire et compressé, et plusieurs flots de données peuvent être multiplexés sur une seule connexion TCP. Comme TCP n'est pas adapté au multiplexage, HTTP/2 peut parfois être très inefficace. Malgré ses problèmes, HTTP/2 est largement déployé aujourd'hui.

HTTP/3

HTTP/3 conserve encore la sémantique de HTTP/1.1, mais n'utilise plus TCP : il est basé sur le protocole de couche de transport QUIC, qui est lui-même basé sur UDP. À la différence de TCP, QUIC intègre le chiffrage au protocole de couche de transport lui-même, et implémente un multiplexage efficace.

Au moment où ce document est écrit, HTTP/3 est implémenté par 70% des navigateurs, mais n'est à ma connaissance déployé du côté serveur que par Google.

Applications web

HTTP a initialement été conçu pour le transfert de documents HTML. Cependant, il a rapidement été utilisé pour implémenter des applications dont l'interface utilisateur s'affiche dans un navigateur web : ce sont les applications web.

L'avantage principal des applications web est qu'elles ne demandent aucune installation de logiciel supplémentaire du côté du client, ce qui est pratique aussi bien pour l'utilisateur que pour le distributeur de l'application.

Génération côté serveur
Les premières applications web étaient implémentées entièrement du côté serveur : un programme (souvent implémenté en PHP) s'exécutait sur le serveur et produisait un document HTML ordinaire, qui était envoyé au navigateur qui le traitait comme une page web. Cette approche a permis de rapidement déployer des applications sans demander aucun changement des clients. Le problème principal de ces applications de première génération était leur faible interactivité : chaque interaction demandait un aller-retour client-serveur et le chargement d'une nouvelle page, ce qui pouvait prendre plusieurs secondes. Malgré cela, certaines applications de première génération sont encore déployées aujourd'hui, notamment Wordpress.

Source : *Poly du cours / Juliusz Chroboczek*

Réponse : le nombre de RTT minimal est 2. Un RTT pour récupérer les ID des messages, et un RTT pour les 50 requêtes de récupération des messages qui s'exécute en parallèle.

Exercice 6.

Modifiez votre programme pour qu'il évite de recevoir une liste où aucun message n'a changé en utilisant l'entête If-None-Match. Pourquoi est-ce préférable à l'entête IfModified-Since?

En-têtes HTTP

- **Last-Modified (réponse)**: L'en-tête **Last-Modified** renseigne sur la date et l'heure de la dernière modification de la page.

- **If-Modified-Since** et **If-Non-Unmodified-Since (requête) :** Les deux en-têtes **If-Modified-Since** et **If-Unmodified-Since** prennent comme valeur une date au format universel, **EEE, dd MMM yyyy HH : mm : ss z** en Java, ou **r** en PHP.

  ```
  GET /fruit/kiwi.gif HTTP / 1.0
  User-agent : Mozilla / 4.0
  If-modified-since : Mon, 22 Jun 1998 09 :23 :24
  ```

  Ce GET conditionnel demande au serveur de n'envoyer l'objet que s'il a été modifié depuis la date de dernière consultation (la valeur de **Last-Modified**). Supposons que ce dernier n'ait pas été modifié depuis le 22 juin 1998 à 09 :23 :24. Alors, le serveur Web envoie au client le message de réponse suivant :

  ```
  HTTP / 1.0 304 Not Modified
  Date : Wed, 19 Aug 1998 15 :39 :29
  Server : Apache / 1.3.0 (Unix)
  ```

  (squelette vide)

  Nous voyons qu'en réponse au GET conditionnel, le serveur Web envoie bel et bien un message de réponse, mais sans inclure l'objet sollicité. Ceci consommerait de la bande passante inutilement et ne ferait qu'augmenter le temps d'attente de l'utilisateur, surtout si l'objet en question est volumineux.

- **Etag (requête [?] et réponse) : Etag** signifie *Entity Tag*. Cet en-tête est positionné dans la réponse. Il permet d'attribuer un identifiant unique (un peu comme une somme de contrôle, mais en mieux. Il peut s'agir d'un

hachage cryptographique) à une ressource. Utilisé principalement dans la gestion de cache, l'Etag varie si la ressource est modifiée.

Etag : "ledec-3e3073913b100"

- **If-Match** et **If-Non-Match (requête):** Les en-têtes **If-Match** et **If-Non-Match** prennent comme valeur une chaine de caractères correspondant au ETag d'une ressource.

  Le client peut valider les copies en cache dont il dispose en envoyant au serveur un en-tête **If-Non-Match** listant les étiquettes des copies en cache. Si l'une des étiquettes correspond au contenu que le serveur s'apprête à envoyer, la copie en cache correspondante peut être utilisée.

  Cette méthode est intéressante quand la détermination de la validité d'une page est ni pratique ni utile. Par exemple, un serveur peut retourner des contenus différents pour la même URL en fonction de la langue et du type MIME préféré par le client. Dans ce cas, la date de modification à elle seule ne suffira pas à permettre au serveur de savoir si la page en cache est encore d'actualité.

Sources :

Analyse structurée des réseaux : des applications de l'internet aux infrastructures de télécommunication */ James Kurose et Keith Ross - 2e édition - DL 2003*

Développer des services REST en Java : Échanger des données au format JSON */ Sobrero Aurélie – 2014*

Réseaux */ Tanenbaum, Andrew - Wetherall, David - 5e édition - DL 2011*

Validateurs HTTP

- **Un validateur** est une étiquette qui est attachée à une donnée et qui permet de vérifier efficacement si deux copies de la donnée sont identiques.
  - La date de dernière modification comme validateur, ce qui n'est pas toujours fiable.
  - Validateurs opaques, les entity tags (**ETag**).

- **Date de dernière modification** : Une réponse HTTP peut contenir la date de dernière modification de la ressource dans l'entête **Last-Modified**.

- **La date de dernière modification n'est pas un validateur fiable**.
  - **Une réponse peut dépendre d'entêtes de la requête** (par exemple l'entête **Accept-Language**), qui font que **deux données ayant la même date de dernière modification peuvent être indépendantes**;

HTTP/1.1 ajoute l'entête Varies, qui permet de déclarer les entêtes dont dépend la réponse.
    - **Les dates HTTP** ont une granularité d'une seconde, et **la date de dernière modification ne permet donc pas de détecter qu'une ressource a changé deux fois durant la même seconde**.

- **ETag** : Dans HTTP/1.1, une donnée peut être étiquetée par le serveur avec un entity tag, un validateur qui est transmis dans l'entête ETag.

  Le ETag est opaque pour le client, qui n'en connaît pas la structure : les seules opérations autorisées sur les ETags sont la comparaison. Comment le serveur génère le ETag est une affaire privée, la seule contrainte est que deux données distinctes aient des ETags distincts.

  Par exemple, le serveur peut utiliser un hash cryptographique de la donnée entière, ce qui est inefficace mais fiable et facile à implémenter.

  Dans les cas où la donnée ne dépend pas de la requête, le serveur peut simplement utiliser la date de dernière modification, qui peut alors être envoyée avec une granularité arbitraire. Selon l'application, le serveur peut aussi utiliser un hash combinant la date de dernière modification avec les entêtes de la requête.

Source :

Poly du cours / Juliusz Chroboczek

Pourquoi est-ce préférable à l'entête If – Modified – Since ?

La date de dernière modification n'est pas un validateur fiable car :

- **Une réponse peut dépendre d'entêtes de la requête** (par exemple l'entête **Accept-Language**), qui font que **deux données ayant la même date de dernière modification peuvent être indépendantes**; HTTP/1.1 ajoute l'entête Varies, qui permet de déclarer les entêtes dont dépend la réponse.
    - Exemple : un serveur peut retourner des contenus différents pour la même URL en fonction de la langue et du type MIME préféré par le client. Dans ce cas, la date de modification à elle seule ne suffira pas à permettre au serveur de savoir si la page en cache est encore d'actualité.

- **Les dates HTTP** ont une granularité d'une seconde, et **la date de dernière modification ne permet donc pas de détecter qu'une ressource a changé deux fois durant la même seconde**.

Sources :

Poly du cours */ Juliusz Chroboczek*

Analyse structurée des réseaux : des applications de l'internet aux infrastructures de télécommunication */ James Kurose et Keith Ross - 2e édition - DL 2003*

Développer des services REST en Java : Échanger des données au format JSON */ Sobrero Aurélie – 2014*

Réseaux */ Tanenbaum, Andrew - Wetherall, David - 5e édition - DL 2011*