

# API avancées

ETag & Rate Limit

Entity Tag et contrôle de version

# Entity Tag (ETag)

Un mécanisme permettant d'identifier la version d'une ressource.

Il est défini par la [RFC 2616, section 3.11](#)

Il sert d'identifiant pour les services de cache et permet donc de limiter l'utilisation de la bande passante.

Il permet aussi de prévenir les modifications simultanées d'une même ressource.

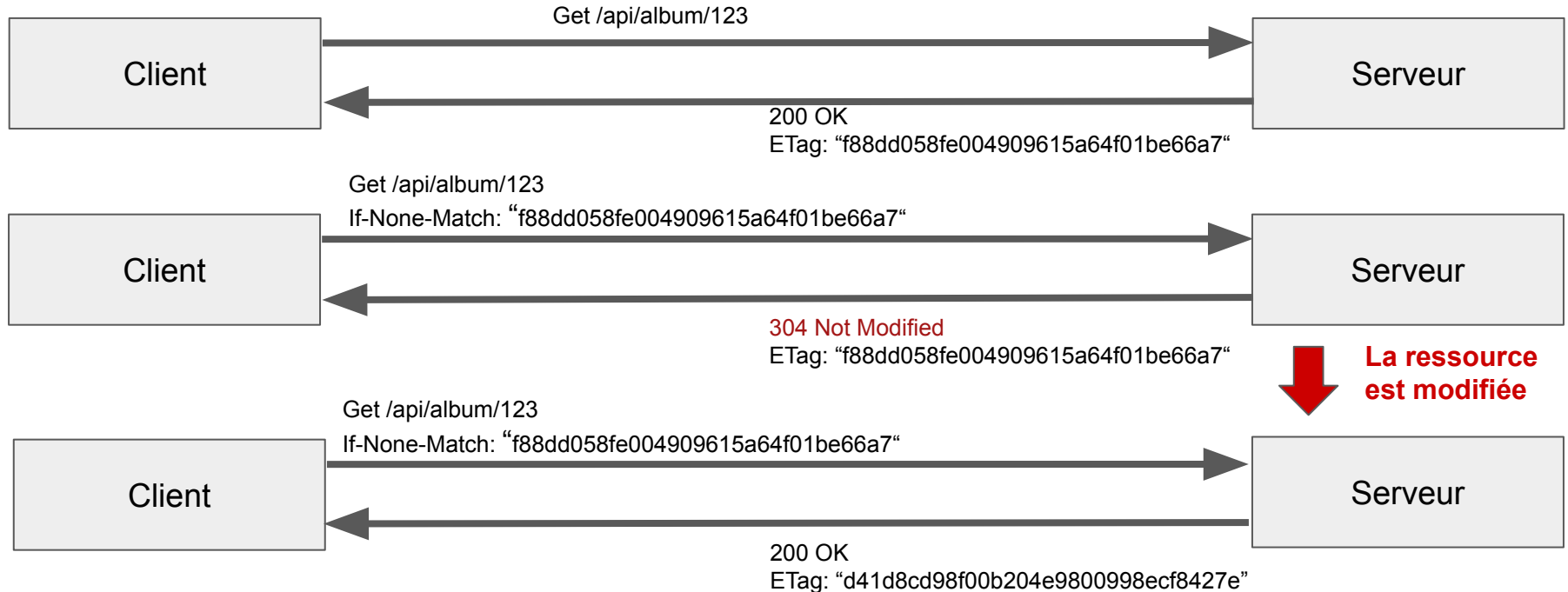
2 ressources différentes doivent avoir des ETag différents. L'ETag est un hash de la ressource.

Fonctionnement :

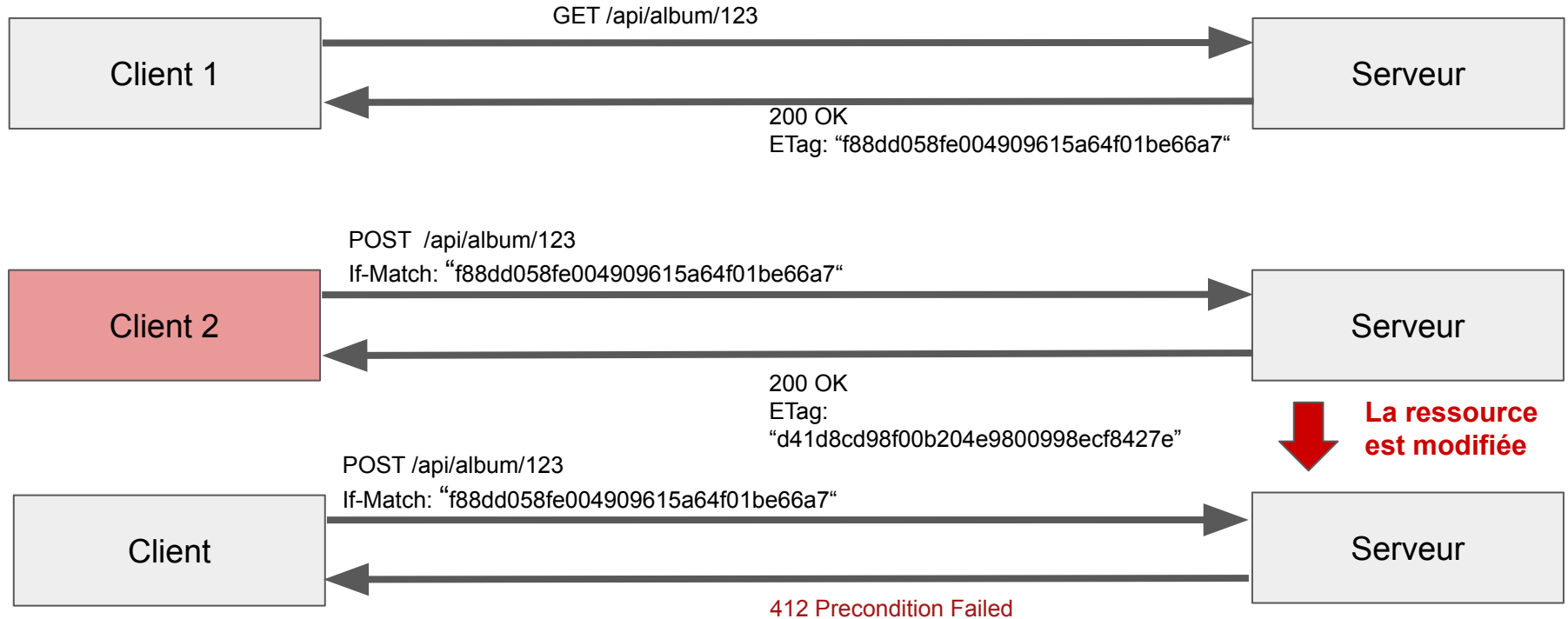
Le serveur communique un HEADER **ETag**: "**hash**" lorsqu'il fournit une ressource

Le client utilise un HEADER **If-None-Match**: "**hash**" ou **If-Match**: "**hash**" lorsqu'il demande une ressource

# Exemple : Réduire l'utilisation de la bande passante



# Exemple : Modifications simultanées



# Strong / weak ETag

Définie dans la [RFC 2616, section 13.3.3](#)

Différencie 2 type de validation (forte et faible)

Forte :

Le validateur (l'ETag) change avec **chaque** changement de la ressource.

Souvent la valeur de hashage de notre entité.

La vérification est au bit près.

Faible :

Le validateur (l'ETag) change seulement lors de **changements sémantiques** de la ressource.

Préfixé par W/

*Last-Modified* est considéré par défaut comme une vérification faible.

# ETag VS Last-Modified

Last-Modified est similaire à ETag bien que moins précis car uniquement basé sur la date de modification de la ressource.

Le client utilise un HEADER ***If-Modified-Since: Date*** ou ***If-Unmodified-Since: Date*** lorsqu'il demande une ressource.

Le serveur lui renvoie un HEADER ***Last-Modified: Date***

ex : Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT

L'Etag est préférable au Last-Modified si une vérification forte est nécessaire (taux de modification supérieur à 1/s)

**La recommandation de la RFC est d'utiliser les 2 si possible**

# Shallow Implementation - Spring

Spring propose ***ShallowEtagHeaderFilter*** pour gérer de manière transparente les ETag.

Mais comme son nom l'indique, il ne gère les ETags qu'en surface. Il calcule l'ETag d'une ressource à partir de la réponse du service, les calculs pour récupérer la ressource sont effectués dans tous les cas.

Ne fonctionne que pour les requêtes GET et la condition IF-NONE-MATCH

ETag sous la forme : 0 + hash MD5 sur 32bits

ex : *077c00bcab66658c79edccafdc17f2186*



# Deep Implementation

Comment faire pour ne pas avoir à recalculer l'ETag à partir de la notre réponse

- S'assurer que nos ressources ne seront pas modifiées sans modification dans le cache

Mais pas de solution simple, cela va dépendre de vos ressources et de votre infrastructure, ...

- Où placer le cache si on a une infrastructure distribué
- Problème de performance si on garde toutes nos ressources dans le cache
- Est-ce que c'est bien nécessaire ?

Rate Limit

# Rate Limit

La limitation du débit consiste à empêcher la fréquence d'une opération de dépasser certaines contraintes.

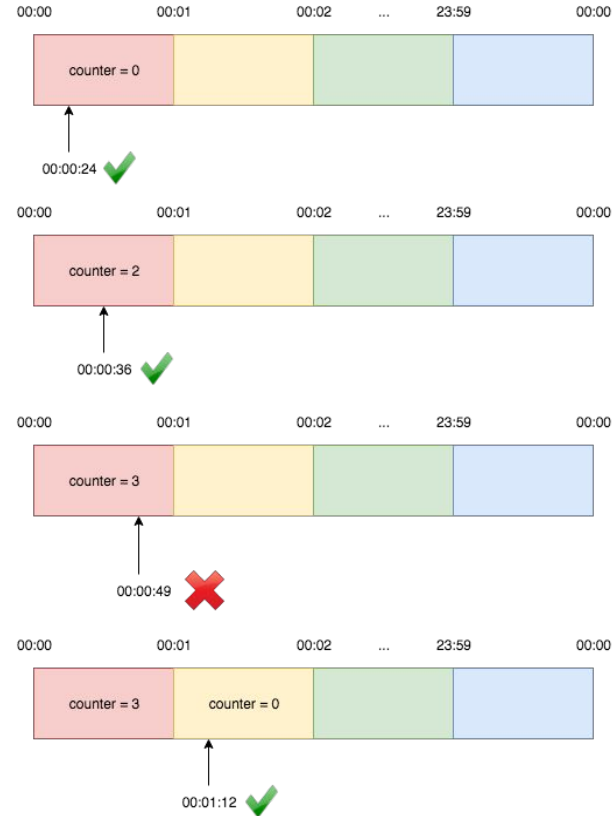
- Se protéger contre une utilisation excessive (intentionnelle ou non) - DoS
- Limiter l'accès en fonction de règles commerciales

Si la limite est dépassée :

- Rejeter la requête → erreur 429 - Too Many Requests
- Mettre dans une file d'attente
- Surcharge financière

# Techniques d'application

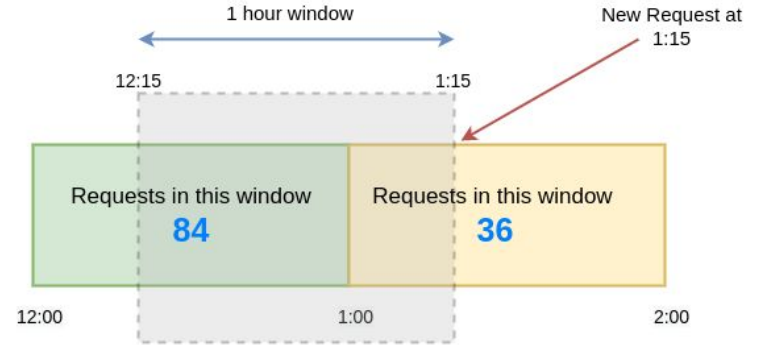
- Fixed window
  - Un nombre maximum de requêtes est fixé pour une fenêtre de temps fixe.
  - Si la limite est dépassée la requête est rejetée, il faudra attendre la prochaine fenêtre
  - Risque de pic de requêtes autour de la jonction entre deux fenêtres.



# Techniques d'application

- Sliding window

- Un nombre maximum de requêtes est fixé pour une fenêtre de temps fixe.
- On calcule une valeur estimée du compteur à un instant  $t$  en pondérant la capacité de la fenêtre précédente avec la fenêtre actuelle.
- Si la valeur estimée dépasse la limite, la requête est rejetée



Limit = 100 requests/hour

<https://systemsdesign.cloud/SystemDesign/RateLimiter>

nlogn.in

ce → Compteur estimé

xp → compteur précédent (84)

tf → taille de la fenêtre (60 minutes)

t → temps passé dans la fenêtre (15)

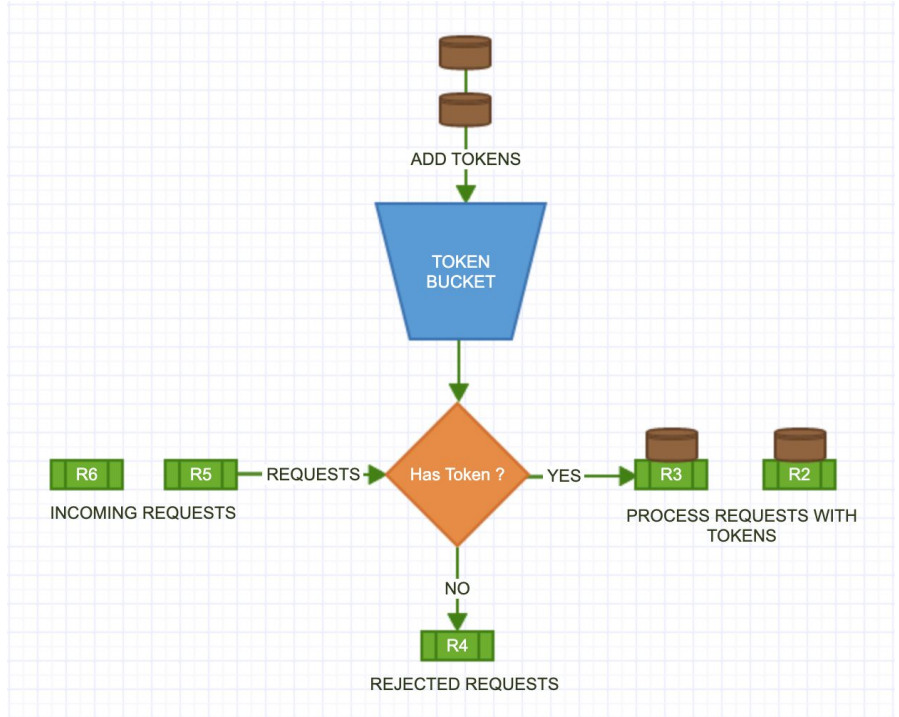
ca → compteur actuelle (36)

$$ce = xp * ((tf - t) / tf) + ca = 84 * 0.75 + 36 = 99$$

ce (99) < limite (100) ⇒ la requête est acceptée

# Techniques d'application

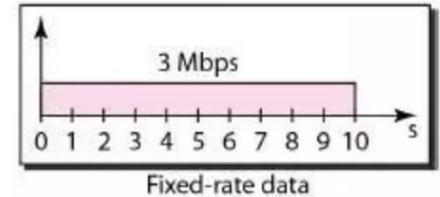
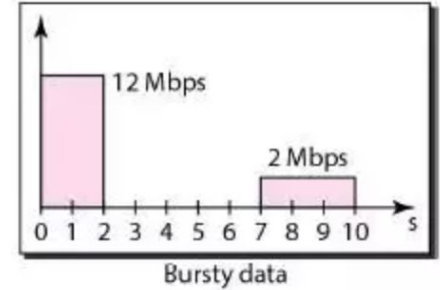
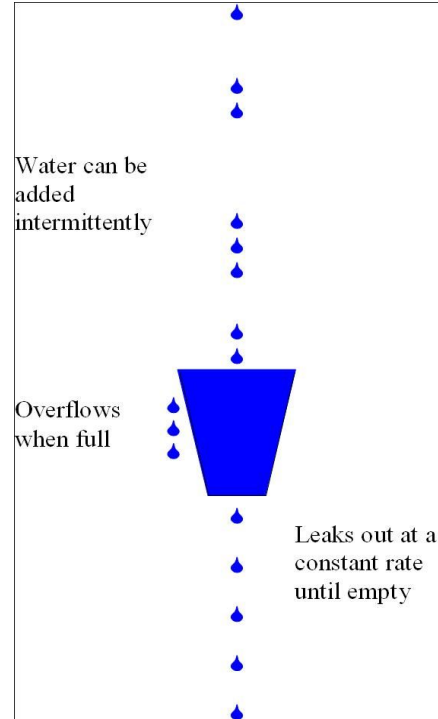
- Token Bucket
  - Le seau d'une capacité fixe se remplit à un rythme constant et chaque requêtes consommes des jetons
  - Si le seau est vide, la requête est rejetée.



# Techniques d'application

- Leaky Bucket

- Le seau d'une capacité fixe se **vide à un rythme constant** (si il n'est pas vide), chaque requête remplit le seau.
- Si le seau est plein, il déborde et la requête est rejetée
- Lisser les pics de consommations
- Le débit de requêtes traité est constant



<https://systemsdesign.cloud/SystemDesign/RateLimiter>

[https://upload.wikimedia.org/wikipedia/commons/c/c4/Leaky\\_bucket\\_analogy.JPG](https://upload.wikimedia.org/wikipedia/commons/c/c4/Leaky_bucket_analogy.JPG)

# Implementation Token Bucket - Bucket4j

Librairie [bucket4j](#) :

- Ajout de jetons toutes les  $r$  seconde dans le seau
- Le sceau à une capacité de  $x$  jetons
- Une requête consomme  $n$  jetons du seau

Terminologie :

- [Bucket](#) → représente le token bucket et fournit les méthodes pour interagir avec
- [Bandwidth](#) → représente la capacité du bucket
- [Refill](#) → représente le taux de remplissage du bucket



# Bucket4j - Terminologie

Méthodes du Bucket :

- ***boolean tryConsume(long numTokens)*** //essaie de consommé x token
- ***ConsumptionProbe tryConsumeAndReturnRemaining(long numTokens)***

Méthodes de la ConsumptionProbe :

- ***boolean isConsumed()*** //donne le résultat de tentative d'utilisation des tokens
- ***long getRemainingTokens()*** // donne le nombre de tokens restant dans le bucket
- ***long getNanosToWaitForRefill()*** // 0 si isConsumed() == true sinon temps en nanosecondes avant que le nombre de tokens nécessaire soit ajouté au bucket

# Bucket4j - configuration

import gradle :

```
implementation group: 'com.github.vladimir-bukhtoyarov', name: 'bucket4j-core', version: '7.6.0'
```

Mise en place

```
//rajoute 10 tokens toutes les minutes
```

```
Refill refill = Refill.intervally(10, Duration.ofMinutes(1));
```

```
//capacité max de 10 token
```

```
Bandwidth limit = Bandwidth.classic(10, refill);
```

```
Bucket bucket = Bucket.builder().addLimit(limit).build();
```

# Bucket4j - Utilisation basique

```
@GetMapping
public ResponseEntity<String> hello() {
    if(bucket.tryConsume(1)) {
        return ResponseEntity.ok("world");
    }
    return ResponseEntity.status(HttpStatus.TOO_MANY_REQUESTS).build();
}
```

# Bucket4j - Informer le client

```
@GetMapping
public ResponseEntity<String> hello() {
    ConsumptionProbe probe = bucket.tryConsumeAndReturnRemaining(1);
    if (probe.isConsumed()) {
        return ResponseEntity.ok()
            .header("X-Rate-Limit-Remaining", Long.toString(probe.getRemainingTokens()))
            .body("world");
    }
    long delaiEnSeconde = probe.getNanosToWaitForRefill() / 1_000_000_000;
    return ResponseEntity.status(HttpStatus.TOO_MANY_REQUESTS)
        .header("X-Rate-Limit-Retry-After-Seconds", String.valueOf(delaiEnSeconde))
        .build();
}
```

# Bucket4j - Refill

- Greedy

- Ajoute des tokens de manière le plus vite possible pour atteindre le max à la fin de l'intervall
  - 10 tokens par 1 seconde, ajoute 1 token toutes les 100 ms

- ```
Refill.greedy(10, Duration.ofSeconds(1));
```

- Interval

- Attend la fin de l'intervall pour générer tous les tokens

- ```
Refill.intervally(100, Duration.ofMinutes(1));
```

- IntervallyAligned

- Comme Interval mais on peut lui configurer l'"Instant" du premier remplissage

- ```
Instant firstRefillTime = ZonedDateTime.now()
```

- ```
.truncatedTo(ChronoUnit.HOURS).plus(1, ChronoUnit.HOURS).toInstant();
```

- ```
Refill.intervallyAligned(400, Duration.ofHours(1), firstRefillTime, true);
```

# Bucket4j - Bandwidth

Une simple Bandwidth utilise un Refill “Greedy”

```
Bandwidth.simple(100, Duration.ofMinutes(1));  
Bandwidth.classic(100, Refill.greedy(100, Duration.ofMinutes(1)));
```

On peut avoir une capacité différente de notre taux de remplissage pour limiter fréquence d'accès à l'API

```
long capaciteMax = 50;  
Refill refill = Refill.greedy(10, Duration.ofSeconds(1));  
Bandwidth limit = Bandwidth.classic(capaciteMax, refill);  
Bucket.builder().addLimit(limit).build();
```

On peut également avoir définir plusieurs Bandwidth

```
Bucket.builder()  
    .addLimit(Bandwidth.simple(1000, Duration.ofMinutes(1))) // 1000 tokens par 1 minute  
    .addLimit(Bandwidth.simple(50, Duration.ofSeconds(1))) // mais pas plus 50 tokens par 1 second  
    .build();
```

# Webographie

<https://apisyouwonthate.com/blog/api-versioning-has-no-right-way>

<https://restfulapi.net/versioning/>

<https://datatracker.ietf.org/doc/html/rfc7231>

<https://www.youtube.com/watch?v=plkA-aPtKNs>

<https://dzone.com/articles/versioning-rest-api-with-spring-boot-and-swagger>

<https://cloud.google.com/architecture/rate-limiting-strategies-techniques>

[https://en.wikipedia.org/wiki/Token\\_bucket](https://en.wikipedia.org/wiki/Token_bucket)

<https://github.com/bucket4j/bucket4j/blob/master/README.md>

<https://systemsdesign.cloud/SystemDesign/RateLimiter>

<https://www.javadevjournal.com/spring/etags-for-rest-with-spring/>

<https://www.youtube.com/watch?v=plkA-aPtKNs>

<https://bucket4j.com/8.1.1/toc.html#quick-start-examples>