jq, traiter du JSON en ligne de commande

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 9 août 2017

https://www.bortzmeyer.org/jq.html

Aujourd'hui, énormément de données sont distribuées dans le format JSON. La simplicité de ce format, normalisé dans le RFC 8259[1], et bien sûr son utilisation dans JavaScript expliquent ce succès. Pour le transport de données (celui du texte est une autre histoire), JSON a largement remplacé XML. Il existe donc des bibliothèques pour traiter le JSON dans tous les langages de programmation. Mais si on veut une solution plus simple, permettant de traiter du JSON depuis la ligne de commande ? C'est là qu'intervient jq, qui permet à la fois de faire des opérations simples en ligne de commande sur des données JSON, et de programmer des opérations complexes, si nécessaires.

jq est parfois présenté comme « l'équivalent de sed pour JSON <https://robots.thoughtbot.com/jq-is-sed-for-json> ». Ce n'est pas faux, puisqu'il permet en effet des opérations simples depuis la ligne de commande. Mais c'est assez court, comme description : jq est bien plus que cela, il inclut même un langage de programmation complet.

Commençons par quelques exemples simples. On va utiliser le JSON produit en utilisant l'outil madonctl <https://github.com/McKael/madonctl> pour récupérer des messages (les pouètes) envoyées sur Mastodon. Si je fais :

```
%  madonctl --output json timeline :afnic
[{"id":3221183,"uri":"tag:mastodon.social,2017-07-20:objectId=13220936:objectType=Status","url":"https://mastodo
ial/users/afnic/updates/3830115","account":{"id":37388,"username":"afnic","acct":"afnic@mastodon.social","displa
e":"Afnic","note":"\u003cp\u003eLe registre des noms de domaine en .fr\u003cbr\u003eRdv sur \u003ca href=\"http:
.afnic.fr/\" rel=\"nofollow noopener\" target=\"_blank\"\u003e   ...
```

J'ai récupéré les pouètes qui parlent de l'AFNIC, sous forme d'une seule ligne de JSON, peu lisible. Alors qu'avec jq :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc8259.txt

```
%  madonctl --output json timeline :afnic | jq .
[
  {
    "id": 3221183,
    "uri": "tag:mastodon.social,2017-07-20:objectId=13220936:objectType=Status",
    "url": "https://mastodon.social/users/afnic/updates/3830115",
    "account": {
      "id": 37388,
      "username": "afnic",
      "acct": "afnic@mastodon.social",
      "display_name": "Afnic",
      "note": "<p>Le registre des noms de domaine en .fr<br>Rdv sur <a href=\"http://www.afnic.fr/\" rel=\"n
oopener\" target=\"_blank\"><span class=\"invisible\">http://www.</span><span class=\"\">afnic.fr/</span><sp
\"invisible\"></span></a></p>",
      "url": "https://mastodon.social/@afnic",
```

J'ai au contraire du beau JSON bien lisible. C'est la première utilisation de jq pour beaucoup d'utilisateurs, dans un monde où les services REST fabriquent en général du JSON très compact (peut-être pour gagner quelques octets), jq peut servir de "*pretty-printer*".

Le point après la commande `jq` est un **filtre**. jq fonctionne par enchainement de filtres et, par défaut, produit des données JSON joliment formatées. (Si, par contre, on ne veut pas de "*pretty-printing*", on lance jq avec l'option `--compact-output`.)

Prenons un exemple où on veut sélectionner une seule information dans un fichier JSON. Mettons qu'on utilise RDAP (RFC 9082) pour trouver de l'information sur une adresse IP. RDAP produit du JSON :

```
% curl -s  http://rdap.db.ripe.net/ip/2001:1470:8000:53::44
{
  "handle" : "2001:1470:8000::/48",
  "startAddress" : "2001:1470:8000::/128",
  "endAddress" : "2001:1470:8000:ffff:ffff:ffff:ffff:ffff/128",
  "ipVersion" : "v6",
  "name" : "ARNES-IPv6-NET",
  "type" : "ASSIGNED",
  "country" : "SI",
...
```

Si on ne veut que le pays du titulaire d'un préfixe d'adresses IP, c'est simple avec jq, avec le filtre `.country` qui veut dire « le champ `country` de l'objet JSON (`.` étant le texte JSON de plus haut niveau, ici un objet) » :

```
% curl -s  http://rdap.db.ripe.net/ip/131.111.150.25 | jq .country
"GB"

% curl -s  http://rdap.db.ripe.net/ip/192.134.1.1 | jq .country
"FR"
```

On n'aurait pas pu utiliser les outils ligne de commande classiques comme grep ou sed car le format JSON n'est pas orienté ligne (et a trop de subtilités pour eux).

Et si on veut interroger son nœud Bitcoin pour connaitre à quel bloc il en est :

```
% bitcoin-cli getinfo | jq .blocks
478280
```

Cet exemple particulier n'est pas super-intéressant, on a déjà `bitcoin-cli getblockcount` mais cela montre bien qu'on peut utiliser jq très simplement, pour des tâches d'administration système.

Au passage, pour trouver le filtre à utiliser, il faut connaitre la structure du fichier JSON. On peut lire la documentation (dans l'exemple RDAP ci-dessus, c'est le RFC 9083) mais c'est parfois un peu compliqué alors que JSON, étant un format texte, fournit une solution plus simple : afficher le début du fichier JSON et repérer les choses qui nous intéressent. (Un format binaire comme CBOR, RFC 8949, ne permet pas cela.) Cette méthode peut sembler du bricolage, mais l'expérience prouve que les services REST n'ont pas toujours de documentation ou, lorsqu'ils en ont, elle est souvent fausse. Donc, savoir explorer du JSON est utile.

Notez que jq produit du JSON, donc les chaînes de caractères sont entre guillemets. Si on veut juste le texte, on utilise l'option `--raw-output` :

```
% curl -s  http://rdap.db.ripe.net/ip/2001:1470:8000:53::44 | jq --raw-output .country
SI
```

Et si on veut un membre d'un objet qui est lui-même un membre de l'objet de plus haut niveau ? On crée un filtre avec les deux clés :

```
%  echo '{"foo": {"zataz": null, "bar": 42}}' | jq .foo.bar
42
```

Un exemple réel d'un tel « double déréférencement » serait avec l'API de GitHub, ici pour afficher tous les messages de "*commit*" :

```
% curl -s https://api.github.com/repos/bortzmeyer/check-soa/commits | \
     jq --raw-output '.[].commit.message'
New rules for the Go linker (see <http://stackoverflow.com/questions/32468283/how-to-contain-space-in-value-stri
Stupid regression to bug #8. Fixed.
Timeout problem with TCP
Query the NN RRset with EDNS. Closes #9
Stupidest bug possible: I forgot to break the loop when we got a reply
TCP support
...
```

Et si le texte JSON était formé d'un tableau et pas d'un objet (rappel : un objet JSON est un dictionnaire) ? jq permet d'obtenir le nième élément d'un tableau (ici, le quatrième, le premier ayant l'indice 0) :

```
% echo '[1, 2, 3, 5, 8]' | jq '.[3]'
5
```

(Il a fallu mettre le filtre entre apostrophes car les crochets sont des caractère spéciaux pour le shell Unix.)

Revenons aux exemples réels. Si on veut le cours du Bitcoin en euros, CoinDesk nous fournit une API REST pour cela. On a juste à faire un triple accès :

```
% curl -s https://api.coindesk.com/v1/bpi/currentprice.json | jq .bpi.EUR.rate
"2,315.7568"
```

Notez que le cours est exprimé sous forme d'une chaîne de caractères, pas d'un nombre, et qu'il n'est pas à la syntaxe correcte d'un nombre JSON. C'est un des inconvénients de JSON : ce format est un tel succès que tout le monde en fait, et souvent de manière incorrecte. (Sinon, regardez une solution sans jq mais avec sed <https://twitter.com/climagic/status/755052827291054080>.)

Maintenant, on veut la liste des comptes Mastodon qui ont pouèté au sujet de l'ANSSI. Le résultat de la requête `madonctl --output json timeline :anssi` est un tableau. On va devoir itérer sur ce tableau, ce qui se fait avec le filtre `[]` (déjà utilisé plus haut dans l'exemple GitHub), et faire un double déréférencement, le membre `account` puis le membre `acct` :

```
% madonctl --output json timeline :anssi | jq '.[].account.acct'
"nschont@mastodon.etalab.gouv.fr"
"ChristopheCazin@mastodon.etalab.gouv.fr"
"barnault@mastodon.etalab.gouv.fr"
...
```

Parfait, on a bien notre résultat. Mais le collègue qui avait demandé ce travail nous dit qu'il faudrait éliminer les doublons, et trier les comptes par ordre alphabétique. Pas de problème, jq dispose d'un grand nombre de fonctions pré-définies, chaînées avec la barre verticale (ce qui ne déroutera pas les utilisateurs Unix) :

```
%  madonctl --output json timeline :anssi | jq '[.[].account.acct] | unique | .[]'
"ChristopheCazin@mastodon.etalab.gouv.fr"
"G4RU"
"Sangokuss@framapiaf.org"
...
```

Oulah, ça devient compliqué ! `unique` se comprend sans difficulté mais c'est quoi, tous ces crochets en plus ? Comme `unique` travaille sur un tableau, il a fallu en fabriquer un : les crochets extérieurs dans l'expression `[.[].account.acct]` disent à jq de faire un tableau avec tous les `.account.acct` extraits. (On peut aussi fabriquer un objet JSON et je rappelle que objet = dictionnaire.) Une fois ce tableau fait, `unique` peut bien travailler mais le résultat sera alors un tableau :

```
%  madonctl --output json timeline :anssi | jq '[.[].account.acct] | unique'
[
  "ChristopheCazin@mastodon.etalab.gouv.fr",
  "G4RU",
  "Sangokuss@framapiaf.org",
...
```

Si on veut « aplatir » ce tableau, et avoir juste une suite de chaînes de caractères, il faut refaire une itération à la fin (le `.[]`).

(Les Unixiens expérimentés savent qu'il existe `uniq` et `sort` comme commandes Unix et qu'on aurait aussi pu faire `jq '.[].account.acct' | sort | uniq`. Chacun ses goûts. Notez aussi qu'il n'y a pas besoin d'un tri explicite en jq : `unique` trie le tableau avant d'éliminer les doublons.)

J'ai dit un peu plus haut que jq pouvait construire des textes JSON structurés comme les tableaux. Ça marche aussi avec les objets, en indiquant la clé et la valeur de chaque membre. Par exemple, si je veux un tableau dont les éléments sont des objets où la clé `href` désigne l'URL d'un pouète Mastodon (ici, les pouètes ayant utilisé le mot-croisillon « slovénie ») :

```
%  madonctl --output json timeline :slovénie | jq "[.[] | { href: .url }]"
[
  {
    "href": "https://mastodon.gougere.fr/users/bortzmeyer/updates/40282"
  },
  {
    "href": "https://mamot.fr/@Nighten_Dushi/3131270"
  },
  {
    "href": "https://mastodon.cx/users/DirectActus/updates/29810"
  }
]
```

Les accolades entourant la deuxième partie du programme jq indiquent de construire un objet, dont on indique comme clé `href` et comme valeur le membre `url` de l'objet original.

Rappelez-vous, jq ne sert pas qu'à des filtrages ultra-simples d'un champ de l'objet JSON. On peut écrire de vrais programmes et il peut donc être préférable de les mettre dans un fichier. (Il existe évidemment un mode Emacs pour éditer plus agréablement ces fichiers source, jq-mode <`https://www.emacswiki.org/emacs/jq-mode`>.) Si le fichier `accounts.jq` contient :

```
# From a Mastodon JSON file, extract the accounts
[.[].account.acct] | unique | .[]
```

Alors, on pourra afficher tous les comptes (on se ressert de `--raw-output` pour ne pas avoir d'inutiles guillemets) :

```
%  madonctl --output json timeline :anssi | jq --raw-output --from-file accounts.jq
ChristopheCazin@mastodon.etalab.gouv.fr
G4RU
Sangokuss@framapiaf.org
...
```

Mais on peut vouloir formater des résultats sous une forme de texte, par exemple pour inclusion dans un message ou un rapport. Reprenons notre nœud Bitcoin et affichons la liste des pairs (Bitcoin est un réseau pair-à-pair), en les classant selon le RTT décroissant. On met ça dans le fichier `bitcoin.jq` :

```
"You have " + (length | tostring) + " peers. From the closest (network-wise) to the furthest away",
       ([.[] | {"address": .addr, "rtt": .pingtime}] | sort_by(.rtt) |
       .[] | ("Peer " + .address + ": " + (.rtt|tostring) + " ms"))
```

Et on peut faire :

```
%   bitcoin-cli getpeerinfo  | jq --raw-output --from-file bitcoin.jq
You have 62 peers. From the closest (network-wise) to the furthest away
Peer 10.105.127.82:38806: 0.00274 ms
Peer 192.0.2.130:8333: 0.003272 ms
Peer [2001:db8:210:5046::2]:33567: 0.014099 ms
...
```

On a utilisé des constructions de tableaux et d'objets, la possibilité de trier sur un critère quelconque (ici la valeur de `rtt`, en paramètre à `sort_by`) et des fonctions utiles pour l'affichage de messages : le signe plus indique la concaténation de chaînes, et la fonction `tostring` transforme un entier en chaîne de caractères (jq ne fait pas de conversion de type implicite).

jq a également des fonctions pré-définies pour faire de l'arithmétique. Utilisons-les pour analyser les résultats de mesures faites par des sondes RIPE Atlas <https://atlas.ripe.net/>. Une fois une mesure lancée (par exemple la mesure de RTT #9211624 <https://atlas.ripe.net/measurements/9211624/>, qui a été créée par la commande `atlas-reach -r 100 -t 1 2001:500:2e::1`, cf. mon article « *"Using RIPE Atlas to Debug Network Connectivity Problems"* <https://labs.ripe.net/Members/stephane_bortzmeyer/using-ripe-atlas-to-debug-network-connectivity-problems> », les résultats peuvent être récupérés sous forme d'un fichier JSON (à l'URL https://atlas.ripe.net/api/v2/me
Cherchons d'abord le RTT maximal :

```
% jq '[.[].result[0].rtt] | max' < 9211624.json
505.52918
```

On a pris le premier élément du tableau car, pour cette mesure, chaque sonde ne faisait qu'un test. Ensuite, on utilise les techniques jq déjà vues, plus la fonction `max` (qui prend comme argument un tableau, il a donc fallu construire un tableau). La sonde la plus lointaine de l'amer <https://www.bortzmeyer.org/amer-mire.html> est donc à plus de 500 millisecondes. Et le RTT minimum ? Essayons la fonction `min`

```
% jq '[.[].result[0].rtt] | min' < 9211624.json
null
```

Ah, zut, certaines sondes n'ont pas réussi et on n'a donc pas de RTT, ce que jq traduit en `null`. Il faut donc éliminer du tableau ces échecs. jq permet de tester une valeur, avec `select`. Par exemple, `(select(. != null)` va tester que la valeur existe. Et une autre fonction jq, `map`, bien connue des programmeurs qui aiment le style fonctionnel, permet d'appliquer une fonction à tous les éléments d'un tableau. Donc, réessayons :

```
% jq '[.[].result[0].rtt] | map(select(. != null)) | min' < 9211624.json
1.227755
```

C'est parfait, une milli-seconde pour la sonde la plus proche. Notez que, comme `map` prend un tableau en entrée et rend un tableau, on aurait aussi pu l'utiliser au début, à la place des crochets :

```
% jq 'map(.result[0].rtt) | map(select(. != null)) | min' < 9211624.json
1.227755
```

Et la moyenne des RTT ? On va utiliser les fonctions `add` (additionner tous les éléments du tableau) et `length` (longueur du tableau) :

```
% jq '[.[].result[0].rtt] | add / length' < 9211624.json
76.49538228
```

(Il y a une faille dans le programme, les valeurs nulles auraient dû être exclues. Je vous laisse modifier le code.) La moyenne n'est pas toujours très utile quand on mesure des RTT. C'est une métrique peu robuste, que quelques "*outliers*" suffisent à perturber. Il est en général bien préférable d'utiliser la médiane <https://www.bortzmeyer.org/mediane-et-moyenne.html>. Une façon simple de la calculer est de trier le tableau et de prendre l'élément du milieu (ou le plus proche du milieu) :

```
% jq '[.[].result[0].rtt] | sort | .[length/2]' < 9211624.json
43.853845
```

On voit que la médiane est bien plus petite que la moyenne, quelques énormes RTT ont en effet tiré la moyenne vers le haut. J'ai un peu triché dans le filtre jq ci-dessus car il ne marche que pour des tableaux de taille paire. Si elle est impaire, `length/2` ne donnera pas un nombre entier et on récupérera `null`. Corrigeons, ce sera l'occasion d'utiliser pour la première fois la structure de contrôle `if`. Notez qu'une expression jq doit toujours renvoyer quelque chose (les utilisateurs de langages fonctionnels comme Haskell ne seront pas surpris par cette règle), donc pas de `if` sans une branche `else` :

```
% jq '[.[].result[0].rtt] | sort | if length % 2 == 0 then .[length/2] else .[(length-1)/2] end' < 9211624.json
43.853845
```

Et maintenant, si on veut la totale, les quatre métriques avec du texte, on mettra ceci dans le fichier source jq. On a utilisé un nouvel opérateur, la virgule, qui permet de lancer plusieurs filtres sur les mêmes données :

```
map(.result[0].rtt) | "Median: " + (sort |
    if length % 2 == 0 then .[length/2] else .[(length-1)/2] end |
    tostring),
                      "Average: " + (map(select(. != null)) | add/length | tostring),
                      "Min: " + (map(select(. != null)) | min | tostring),
                      "Max: " + (max | tostring)
```

Et cela nous donne :

```
% jq --raw-output --from-file atlas-rtt.jq < 9211624.json
Median: 43.853845
Average: 77.26806290909092
Min: 1.227755
Max: 505.52918
```

Peut-on passer des arguments à un programme jq ? Évidemment. Voyons un exemple avec la base des « espaces blancs » décrite dans le RFC 7545. On va chercher la puissance d'émission maximale autorisée pour une fréquence donnée, avec ce script jq, qui contient la variable `frequency`, et qui cherche une plage de fréquences (entre `startHz` et `stopHz`) comprenant cette fréquence :

```
.result.spectrumSchedules[0].spectra[0].frequencyRanges[] |
     select (.startHz <= ($frequency|tonumber) and .stopHz >= ($frequency|tonumber)) |
     .maxPowerDBm
```

On l'appelle en passant la fréquence où on souhaite émettre :

```
% jq --from-file paws.jq --arg frequency  5.2E8  paws-chicago.json
15.99999928972511
```

(Ne l'utilisez pas telle quelle : par principe, les espaces blancs ne sont pas les mêmes partout. Celui-ci était pour Chicago en juillet 2017.)

Il est souvent utile, quand on joue avec des données, de les grouper par une certaine caractéristique, par exemple pour compter le nombre d'occurrences d'un certain phénomène, ou bien pour classer. C'est le classique `GROUP BY` de SQL, qui a son équivalent en jq. Revenons à la liste des comptes Mastodon qui ont pouèté sur l'ANSSI et affichons combien chacun a émis de pouètes. On va grouper les pouètes par compte, fabriquer un objet JSON dont les clés sont le compte, le trier, puis afficher le résultat. Avec ce code jq :

```
# Count the number of occurrences
[.[].account] | group_by(.acct) |
                # Create an array of objects {account, number}
                [.[] | {"account": .[0].acct, "number": length}] |
                # Now sort
                sort_by(.number) | reverse |
                # Now, display
                .[] | .account + ": " + (.number | tostring)
```

On obtiendra :

```
% madonctl --output json timeline :anssi | jq -r -f   anssi.jq
gapz@mstdn.fr: 4
alatitude77@mastodon.social: 4
nschont@mastodon.etalab.gouv.fr: 2
zorglubu@framapiaf.org: 1
sb_51_@mastodon.xyz: 1
```

Voilà, on a bien avancé et je n'ai toujours pas donné d'exemple avec le DNS. Un autre cas d'utilisation de `select` va y pourvoir. Le DNS Looking Glass <`https://www.bortzmeyer.org/dns-lg-usage.html`> peut produire des résultats en JSON, par exemple ici le SOA du TLD `.xxx` : `curl -s -H 'Accept: application/json' https://dns.bortzmeyer.org/xxx/SOA`. Si je veux juste le numéro de série de cet enregistrement SOA ?

```
% curl -s -H 'Accept: application/json' https://dns.bortzmeyer.org/xxx/SOA | \
    jq '.AnswerSection[0].Serial'
2011210193
```

Mais il y a un piège. En prenant le premier élément de la "*Answer Section*", j'ai supposé qu'il s'agissait bien du SOA demandé. Mais l'ordre des éléments dans les sections DNS n'est pas défini. Par exemple, si une signature DNSSEC est renvoyée, elle sera peut-être le premier élément. Il faut donc être plus subtil, et utiliser `select` pour ne garder que la réponse de type SOA :

```
% curl -s -H 'Accept: application/json' https://dns.bortzmeyer.org/xxx/SOA | \
    jq '.AnswerSection | map(select(.Type=="SOA")) | .[0].Serial'
2011210193
```

Notre code jq est désormais plus robuste. Ainsi, sur un nom qui a plusieurs adresses IP, on peut ne tester que celles de type IPv6, ignorant les autres, ainsi que les autres enregistrements que le serveur DNS a pu renvoyer :

```
% curl -s -H 'Accept: application/json' https://dns.bortzmeyer.org/gmail.com/ADDR | \
    jq '.AnswerSection | map(select(.Type=="AAAA")) | .[] | .Address'
"2607:f8b0:4006:810::2005"
```

Bon, on approche de la fin, vous devez être fatigué·e·s, mais encore deux exemples, pour illustrer des trucs et concepts utiles. D'abord, le cas où une clé dans un objet JSON n'a pas la syntaxe d'un identificateur jq. C'est le cas du JSON produit par le compilateur Solidity. Si je compile ainsi :

```
% solc  --combined-json=abi registry.sol > abi-registry.json
```

Le JSON produit a deux défauts. Le premier est que certains utilisateurs ont une syntaxe inhabituelle (des points dans la clé, et le caractère deux-points) :

```
{"registry.sol:Registry": {"abi": ...
```

jq n'accepte pas cette clé comme filtre :

```
%  jq '.contracts.registry.sol:Registry.abi' abi-registry.json
jq: error: syntax error, unexpected ':', expecting $end (Unix shell quoting issues?) at <top-level>, line 1:
.contracts.registry.sol:Registry.abi
jq: 1 compile error
```

Corrigeons cela en mettant la clé bizarre entre guillemets :

```
% jq '.contracts."registry.sol:Registry".abi' abi-registry.json
```

On récupère ainsi l'ABI du contrat. Mais, et là c'est gravement nul de la part du compilateur, l'ABI est une chaîne de caractères à évaluer pour en faire du vrai JSON ! Heureusement, jq a tout ce qu'il faut pour cette évaluation, avec la fonction `fromjson` :

```
% jq '.contracts."registry.sol:Registry".abi | fromjson' abi-registry.json
```

Ce problème des clés qui ne sont pas des identificateurs à la syntaxe traditionnelle se pose aussi si l'auteur du JSON a choisi, comme la norme le lui permet, d'utiliser des caractères non-ASCII pour les identificateurs. Prenons par exemple le fichier JSON des verbes conjugués du français, en . Le JSON est :

```
[{ "Indicatif" : { "Présent" : [ "abade", "abades", "abade", "abadons", "abadez", "abadent" ], "Passé compos
```

Il faut donc mettre les clés non-ASCII entre guillemets. Ici, la conjugaison du verbe « reposer » :

```
% jq 'map(select(.Infinitif."Présent"[0] == "reposer"))' verbs.json
[
  {
    "Indicatif": {
      "Présent": [
        "repose",
        "reposes",
        "repose",
   ...
```

Notez toutefois que jq met tout en mémoire. Cela peut l'empêcher de lire de gros fichiers :

```
% ls -lh la-crime.json
-rw-r--r-- 1 stephane stephane 798M Sep  9 19:09 la-crime.json

% jq .data la-crime.json
error: cannot allocate memory
zsh: abort      jq .data la-crime.json
```

Et un dernier exemple, pour montrer les manipulations de date en jq, ainsi que la définition de fonctions. On va utiliser l'API d'Icinga pour récupérer l'activité de la supervision. La commande `curl -k -s -u operator:XXXXXXX -H 'Accept: application/json' -X POST 'https://ADRESSE:PORT/v1/` va nous donner les changements d'état des machines et des services supervisés. Le résultat de cette commande est du JSON : on souhaite maintenant le formater de manière plus compacte et lisible. Un des membres JSON est une date écrite sous forme d'un nombre de secondes depuis une "*epoch*". On va donc la convertir en jolie date avec la fonction `todate`. Ensuite, les états Icinga (`.state`) sont affichés par des nombres (0 = OK, 1 = Avertissement, etc.), ce qui n'est pas très agréable. On les convertit en chaînes de caractères, et mettre cette conversion dans une fonction, `s2n` :

```
def s2n(s):
  if s==0 then "OK" else if s==1 then "Warning" else "Critical" end  end;
(.timestamp | todate) + " " + .host + " " + .service + " " +  " " + s2n(.state) +
    " " + .check_result.output
```

Avec ce code, on peut désormais exécuter :

```
% curl -k -s -u operator:XXXXXXX -H 'Accept: application/json' -X POST 'https://ADRESSE:PORT/v1/events?types=Sta
   jq --raw-output --unbuffered --from-file logicinga.jq
...
2017-08-08T19:12:47Z server1 example Critical HTTP CRITICAL: HTTP/1.1 200 OK - string 'Welcome' not found on 'ht
```

Quelques lectures pour aller plus loin :

— La page officielle de jq <https://stedolan.github.io/jq/>, avec son excellent manuel <https://stedolan.github.io/jq/manual/>, une bonne FAQ <https://github.com/stedolan/jq/wiki/FAQ> et plein d'exemples d'utilisation de jq <https://github.com/stedolan/jq/wiki/Cookbook>.
— En parlant de manuel, la "*man page*" de jq est très complète.
— On trouve aussi un tutoriel officiel <https://stedolan.github.io/jq/tutorial/> mais très court, c'est juste un début d'introduction. Je préfère le tutoriel « "*Reshaping JSON with jq*" <https://programminghistorian.org/lessons/json-and-jq> », avec ses exemples tirés du monde des musées.
— Autre application pratique de jq, pour la cartographie, un tutoriel par l'exemple <http://shaarli.guiguishow.info/?KMmDNA>.
— Il existe un logiciel proche de jq mais où le langage de programmation est Python, pjy <https://pypi.python.org/pypi/pjy>.
— Un autre système spécialisé pour traiter du JSON est le langage JMESPath <http://jmespath.org/>. Il dispose d'une mise en œuvre en ligne de commande, jp.
— Et une troisième alternative à jq est jshon <http://kmkeen.com/jshon/>.
— Un article en français sur l'utilisation de jq pour fouiller dans les données Twitter : l'analyse du hashtag #TelAvivSurSeine <http://ksahnine.github.io/datascience/unix/bigdata/2015/08/14/analyse-hashtag-telavivsurseine.html>.
— Et bien sûr comme d'habitude l'incontournable StackOverflow <https://stackoverflow.com/questions/tagged/jq>.

Merci à Jean-Edouard Babin et Manuel Pégourié-Gonnard pour la correction d'une grosse bogue dans la première version de cet article.