./jq – Swiss Army Knife of JSON

Markus Geiger <mg@evolution515.net>

Protean Linux | Cloud | DevOps Engineer

Meetup Edition | Full Edition

last update: 2023-11-21 10:00

Press ? for help!

How Do We Query JSON?

A list of VM Instances

```
{
  "Reservations": [
    {
      "Groups": [],
      "Instances": [
        {
          "AmiLaunchIndex": 0,
          "ImageId": "ami-7c82c36a",
          "InstanceId": "i-0d1b0b067617fe29c",
          "InstanceType": "t2.medium",
          "KeyName": "johndoe@macbook",
          "PrivateDnsName": "ip-10-1-4-67.ec2.internal
          "PrivateIpAddress": "10.1.4.67",
          "ProductCodes": [],
          "PublicDnsName": "ec2-34-198-239-245.compute
          "PublicIpAddress": "34.198.239.245",
          "…": "…"
        }, {
          "AmiLaunchIndex": 0,
          "ImageId": "ami-7c82c36a",
          "InstanceId": "i-0e52defe4897b1529",
          "InstanceType": "t2.micro",
```

⇒ Let's get only IDs and Private IP addresses in a list!

JSONPath

Query expressions for JSON (based on “Xpath”)

```
$.Reservations[*].Instances[*].InstanceId
$.Reservations[*].Instances[*].PrivateIpAddress
```

OUTPUT

```
[
  "i-0d1b0b067617fe29c",
  "i-0e52defe4897b1529"
]
[
  "10.1.4.67c",
  "10.1.4.75"
]
```

JMESPath

Fast querying and simple transformation

```
aws ec2 describe-instances \
    --query 'Reservations[*].Instances[*]|[InstanceId,
    --output text
```

OUTPUT

```
i-0d1b0b067617fe29c 10.1.4.67
i-0e52defe4897b1529 10.1.4.75
```

JQ

Looks similar?

```
aws ec2 describe-instances \
   | jq '.Reservations[].Instances[]|[.InstanceId, .Pri
```

OUTPUT

```
i-0d1b0b067617fe29c 10.1.4.67
i-0e52defe4897b1529 10.1.4.75
```

JQ Does More...

JQ is a Programming Language

⇒ fully-featured functional programming language

⇒ uses JSON (indeed BISON) as underlying data type

JQ is a Command Line Utility

⇒ therefore can be part of any CLI toolchain

Finally, *JQ is Back!*

Version 1.7 was released this year.

JQ is Turing Complete

Definition

Turing Complete

> *Solve any problem that can be described and executed in an algorithmic form.*

⇒ Imagine mathematical formulas on endless paper

⇒ *Alan Turing did similar things with his computers in the datacenter “The Turing Machine” (math model) came into existence*

Proof

⇒ Brainfuck Interpreter (GitHub) [1]

⇒ jqjq – JQ implemented in JQ (GitHub)

⇒ JSON Formatter in JQ (medium.com)

⇒ JMESPath implemented in JQ hasn't been done, but should be possible!

⇒ Games, CLI Utilities, ...? *Can be done!*

JQ vs. JMESPath

JQ is Hard to Learn?!

JMESPATH

```
aws ec2 describe-images \
   --owners amazon --region=us-east-1 \
   --filters "Name=architecture,Values=arm64"  "Name=na
   --query 'sort_by((Images[*].[CreationDate,ImageId,Im
   --output table
```

JQ

```
aws ec2 describe-images \
     --owners amazon --region=us-east-1 \
     --filters "Name=architecture,Values=arm64"  "Name=
    | jq -er '[.Images[]|[.CreationDate,.ImageId,.Image
```

➔ It depends on what you do with JQ

➔ It depends on what you do with JQ

➔ Does it fit your requirements?

Language Comparision

Now some slides for the *compiler geeks* ...

JQ Language

jq is a very high-level lexically scoped functional programming language

⇒ in which every JSON value is a constant.

⇒ jq supports backtracking and managing indefinitely long streams of JSON data

⇒ `./jq` is is related to Icon and Haskell programming languages.

JQ "PEG" based on GNU BISON

There is a very close relationship between jq and the parsing expression grammar (PEG) formalism.

JQ shares the equivalence of the seven basic PEG operations shown in the following table:

PEG operation name	PEG notation	jq operation or def
Sequence	`e1 e2`	`e1
Ordered choice	`e1 / e2`	`e1 // e2`
Zero-or-more	`e*`	`def star(E): (E
One-or-more	`e+`	`def plus(E): E
Optional	`e?`	`def optional(E): E // .;`
And-predicate	`&e`	`def amp(E): . as $in
Not-predicate	`!e`	`def neg(E): select( [E] == [] );`

⇒ Read Parsing Expression Grammars: jq as a PEG engine

JMESPath ABNF (Augmented Backus–Naur form)

> *"JQ doesn’t have a spec, we can guarantee that each implementation works the same!"*
>
> ```
> expression        = sub-expression / index-expression  / com
> expression        =/ or-expression / identifier
> expression        =/ and-expression / not-expression / paren-expr
> expression        =/ "*" / multi-select-list / multi-sel
> expression        =/ function-expression / pipe-express
> expression        =/ current-node
> sub-expression    = expression "." ( identifier /
>                                      multi-select-list /
>                                      multi-select-hash /
>                                      function-expression /
>                                      "*" )
> ```
>
> *https://jmespath.org/specification.html*

But only the a 3rd party implementation of JMESPath do use ANTLR!

JAQ – *even more formalism!*

Michael Färber write Denotational Semantics and a Fast Interpreter for jq (https://arxiv.org/abs/2302.10576) for his jaq JQ interpreter:

> *…it's [jq] semantics are currently only specified by its implementation, making it difficult to reason about its behavior […]*
>
> *I provide a syntax and denotational semantics for a subset of the jq language. In particular, the semantics provide a new way to interpret updates.*
>
> *I implement an extended version of the semantics in a novel interpreter for the jq language called jaq. Although **jaq** uses a significantly simpler approach to execute jq programs than jq, jaq is faster than jq on ten out of thirteen benchmarks.*

⇒ jaq 🖤 Rust

⇒ JQJQ is also good for learning PEG

JQ Usage Examples

Pretty Printing

INPUT

```
# assume input.json is pretty big
# JQ will do a great and very fast job!
```

```
$ jq . input.json
$ cat input.json  | jq
```

OUTPUT

```
{
  "you": [
    "are",
    {
      "so": "pretty",
      "big": "data:application/binary;IlN0YXRlVHJhbnNpd
    }
  ]
}
```

Compacting

INPUT

```
{
  "you": [
      { "are": "such" },
    { "a": "pretty" },
    "boy"
  ]
}
```

SHELL

```
# let's also delete .you.are
$ cat input | jq delete(.you.are)-c
```

OUTPUT

```
{"you":[{"a":"pretty"},"boy"]}
```

Getting values from REST APIs

Using cURL or any other CLI

```
$ curl -sSfL  'https://wttr.in/~munich?format=j1' \
    | jq -r '.current_condition[0].FeelsLikeC'
9
```

Posting to REST APIs

You can also construct JSON!

```
# note we don't need "" around key
$ jq -rnc --arg foo foo '{a: {b: $foo}}'
{"a":{"b":"1"}}

# short version (you don't need to initialize b first!)
$ jq -rnc .b.c.d=1
{"b":{"c":{"d":1}}}
```

Posting to REST APIs

You can also construct JSON!

```
# and you could use that in curl
MESSAGE="This should get correctly encoded as 'JSON Str
SENDER="me"
curl -fL \
    -X POST -H "Content-Type: application/json" \
    -d "$( jq -nc \
            --arg "message" "$MESSAGE" \
            --arg sender "$SENDER" '.sender=$SENDER|.me
        )" \
    https://www.timecapsules.space | jq -e
```

Escaping URLs

```
# You can use builtin functions
$ jq -Rr @uri <<< "foo/bar/ÄÖÜ"
foo%2Fbar%2F%C3%84%C3%96%C3%9C
```

Better Grep [1]

[^1]: using `oniguruma` Regex engine

INPUT

```
foo
bar
foobar
foobaz
```

CLI

```
$ cat input| jq -Rre 'select(test("^foo(?!bar)"))'
foo
foobaz
```

Extract AWS ARNs from Terraform State

```
# Extract all strings which begin with arn: or stuff li
$ terraform pull state \
  | jq -re '..|strings|select(test("^arn:|^[a-z]+-[a-f0
```

Stream Filter

INPUT

```
2023-11-11 11:11:11 ERROR unexpected error with response {"errorMessage": "with-some-json",
  "errorCode": -1, sourceHost: "foo.bar.example.com"}
2023-11-11 11:11:11 ERROR unexpected error with response {"errorMessage": "with-some-json",
  "errorCode": -1, sourceHost: "fazbaz.example.com"}
```

CLI

```
# we turn this line into JSON and filter for every hosts which starts with 'foo
$ cat log | jq -rRs '
   select(test("ERROR"))
   |capture("(?<ts>\\d{4}/\\d{2}/\\d{2}.+\\s\\d{2}:\\d{2}:\\d{2}).+with respons
   |select(.json?|fromjson|.sourceHost?|test("^(foo|bar)"))
   |[.ts, .sourceHost ,.errorMessage]
'
```

OUTPUT

```
2023-11-11 11:11:11 foo.bar.example.com with-some-json
```

Transformation

INPUT

```
{
  "Tags": [
    {"Key:" "foo", "Value": "bar"},
    {"Key:" "agent", "Value": "smith"}
  ]
}
```

QUERY

```
.Tags = ((.Tags? // [] | map({(.Key):.Value|tostring})
```

OUTPUT

```
{ "Tags": {"foo": "bar", "agent": "smith"} }
```

Transformation

INPUT

```
{
  "Tags": [
    {"Key:" "foo", "Value": "bar"},
    {"Key:" "agent", "Value": "smith"}
  ]
}
```

REFACTOR

```
def aws_map_tags:
    ((.? // [] | map({(.Key):.Value|tostring}) | add) /

.Tags=(.Tags|aws_map_tags)

# can be written shorter by
.Tags|=aws_map_tags
```

OUTPUT

```
{ "Tags": {"foo": "bar", "agent": "smith"} }
```

Syntax Checking embedded scripts in YAML

```
oq -re -iyaml '.[].script' .gitlab-ci.yml | bash -n
```

More Functions? Implement yourself!

```
# Markdown Table output for arrays
# $ jq -rnc '[{"a": 1}, {"b": 2}]|tabelize|md'
# | a | b |
# |---|---|
# | 1 | - |
# | - | 2 |
def md:
    "| " + (.[0]|join(" | ")) + " |",
    "|-" + (.[0]|map("-")|join("-|-")) + "-|",
    (.[1:][]|"| " + (.|join(" | ")) + " |");
```

Check out my `[~/.jq](https://gitlab.com/-/snippets/3620846/raw/main/.jq)` library!

Sapient Intelligence

Primary Resources to Ingest

JQ Manual | JQ Play | JQ Cookbook | JQ Wiki | FAQ | JBOL | Rosetta Code on JQ

Working with JSON

JSONL “Line Processing”

Unix Pipelines

```
$ printf "1\n2\n3" | cat
5
9
13
```

Do you do JSON Logging?

```
{level: "ERROR", message: "Out of pizza", source: "api"
{level: "ERROR", message: "Out of coke", source: "api",
```

```
my-api-client get-logs | jq 'select(.userId=="2")|[.tim
```

OUTPUT

```
2077-06-11 11:11:11 ERROR Out of coke
```

JSON Supersets

By converting back to these original formats you will loose information!

⇒ YAML *current: 1.2.2 (2021-10-01))*
  ⇒ v1.1: cannot escape forward slash `/`
  ⇒ v1.2: anchors, references, functions, ...

⇒ JSON5 HanSON *JSON for Humans with comments*

⇒ TOML *best of INI and JSON with semantics*

⇒ Hashicorp HCL - Human and Machine friendly Configuration language

⇒ *and others*

No Need to Convert to JSON!

Your CLI tools might be already support JSON output!

⇒ `kubectl -ojson or -ojsonpath`

⇒ `docker --format=json`

⇒ `helm -o json`

⇒ Unix tools `lsblk -J` or if not supported JC

And then you could use jq custom function `toyaml` to write YAML. Or use a wrapper.

CLI Options

But before we head to some grammer, let's look on some JQ – and not only CLI – options.

`-n` null input

With *null input* you don't need to provide stdin

```
$ jq -n .foo.bar=12
{
  "foo": {
    "bar": 12
  }
}
```

`-r` RAW output

```
# Default jq output
$ echo '{"foo":"bar"}' | jq .foo
"bar"

# If it's a string, don't output quotes
$ echo '{"foo":"bar"}' | jq -r .foo
bar

# For JSON it's not changing anything
$ echo '{"foo":"bar"}' | jq -r .
{ "foo": "bar"}
```

`-R` RAW input

This creates the possibility to process plain text

```
$ printf 'foo\nbar' | jq -R .
"foo"
"bar"
```

-s slurp

```
$ printf '{"line":1}\n{"line":2}' | jq -c
{"line":1}
{"line":2}

$ printf '{"line":1}\n{"line":2}' | jq -sc
[{"line":1},{"line":2}]

# combine with raw input
$ printf 'foo\nbar' | jq -Rs .
"foo\nbar"

# we need --slurp in order avaid line-by-line processin
$ printf 'foo\nbar' | jq -Rsc 'split("\n")'
["foo","bar"]
```

-e Exit on error

You can use JQ to evaluate JSON/JQ expressions

```
$ jq -ne 'true,false'; printf "exitcode=$?"
true
false
exitcode=1

$ jq -ne 'empty' &>/dev/null; printf "exitcode=$?"
exitcode=4

$ jq -ne '';  printf "exitcode=$?"
null
exitcode=1
```

Grammar

JQ Builtins

> *Sometime Implemented in JQ itself – list by `builtins` or even overwrite!*
> *https://github.com/jqlang/jq/blob/master/src/builtin.jq*

⇒ **Control Structures** `if` `then` `else` `try` `catch` `break` `while` `until` `foreach` `reduce` `label`

⇒ **Module System** `import` `include` `module` `modulemeta`

⇒ **Functional Patterns** `all` `any` `map` `flatten` `add` `length` `reduce`

⇒ **Path Functions** *(to modify big object)* `path` `paths` `getpath` `setpath`

⇒ **Debug Functions**

`builtins` `stderr` `debug` `error`, `halt`, `halt_error …`

⇒ ***And a lot of other (mostly) useful functions!***

JQ Pipelines

```
# we start with an array and iterate over it by '[]'
["Hel"][]|debug
# current value is now "Hel"

# We assign "Limbo" to variable $who
|"Limbo" as $who|debug
# current value is still "Hel"

# We assign current valie + "lo" to variable $greet
|(.+="lo") as $greet|debug
# current value is "Hel"

# We define a new array with the current value is first
|[$greet, $who]|debug
# current value is now is ["Hello","Limbo"]

# And we pipe it into join function and add "!"
|join(", ")+"!"|debug
# current value is now is "Hello Limbo!"
```

Selection

Lets select only `2` from the input array which is [1,2,3]

```
$ jq -ncr '[1,2,3][]|if (.==2) then .*=10 else . end'
1
20
3
```

Now let’s replace our else case with `empty` as result

```
$ jq -ncr '[1,2,3][]|if (.==2) then . else empty end'
2
```

`empty` is terminates the line.

`select(condition)`

Some behavior can also be accomplished by using `select()` function

```
$ jq -ncr '[1,2,3][]|select(.==2)'
2
```

Attention: Variable assignment

```
$ jq -nc '"foo"|select(false) as $x|.'
# returns nothing

$ jq -nc '"foo"|select(false) // true as $x|.'
"foo"
```

Since v1.7 we now also have "if>then>end" without else, but if if does not apply it passes through the valiue

```
$ jq-1.7 -ncr '[1,2,3][]|if (.==2) then .*=10 end'
1
20
3
```

Line Duplication

Object Duplication

One Object can become three objects!

```
#!/usr/bin/env -S sh -c 'exec jq -nc $0'
[1,2,3] as $example|empty,

{message: "foo", num: $dupe[]}
```

```
$ jq -nc '|'
{"message":"foo","num":1}
{"message":"foo","num":2}
{"message":"foo","num":3}
```

Line Duplication

Object Duplication

Let's assume you want to select keys for a row.

```
#!/usr/bin/env -S sh -c 'exec jq -rcnf $0'
[
    {line:"one",tags:[{key:"foo"}]},
    {line:"two",tags:[{key:"bar"},{key:"baz"}]}
] as $example|empty,

$example[]|{line:.line, onetag: (.tags[].key)}
```

OUTPUT

```
{"line":"one","onetag":"foo"}
{"line":"two","onetag":"bar"}
{"line":"two","onetag":"baz"}
```

Now this could happen by easily by accident!

Elements in Arrays just will be inserted

Now what happens with arrays? The elements are just inserted:

```
#!/usr/bin/env -S sh -c 'exec jq -rcnf $0'
[1,2,3] as $example|empty,

["foo","bar",$dupe[]]
```

```
[1,2,1,2,3]
```

We end up with a second row with suddenly 3 elements!

Error Suppression / Optional Operator: `?`

The `?` operator, used as `EXP?`, is shorthand for `try EXP`.

```
# define input array wih three entries
[{}, true, {"a":1}] as $input|empty,

$input|[.[] | .a]
```

```
jq: error (at <unknown>): Cannot index boolean with str
exit status 5
```

Error Suppression / Optional Operator: `?`

The `?` operator, used as `EXP?`, is shorthand for `try EXP`.

```
# define input array wih three entries
[{}, true, {"a":1}] as $input|empty,

$input|[.[] | .a?],
$input|[.[] | try .a],
$input|[.[] | try .a catch empty],
$input|[.[] | .a? // null]
```

```
[null,1]
[null,1]
[null,1]
[null,null,1]
[null,null,1]
```

Formats

These will always result in a `string`

- ⇒ `@json` Serialize as JSON
- ⇒ `@text` Convert to String
- ⇒ `@html` Escape for HTML
- ⇒ `@url` Encode for use in URL
- ⇒ `@csv/@tsv` Convert flat array into CSV
- ⇒ ...or write your own

(TODO) Recursing and Path Functions

`..` splits up complex data types and can be used to quickyl identify objects within a nested structure

```
{
    foo: [
        "hey",
        {that: {pattern: {"message": "yeah"}}},
        {bar: {that: {"pattern": "not"}}}
    ]
} as $example|empty,

$example|..|objects|select(.that.pattern?|objects)
```

```
{"that":{"pattern":{"message":"yeah"}}}
```

SQL*-like* Functions

INDEX

```
jq -nc '[{a:1},{a:2},{a:3}]|INDEX(.[];.a)'
{"1":{"a":1},"2":{"a":2},"3":{"a":3}}
```

JOIN

```
jq -nc '([{id:1, foo:1},{id:3, foo:3},{id:10, foo:2}])
{"id":1,"bar":1,"foo":1}
{"id":6,"bar":3}
{"id":10,"bar":2,"foo":2}
```

Debugging

JQ doesn't have an interactive debugger – *yet!*

```
# if we hit n==2 then we exit with an error
$ jq -ncr '[1,2,3][]|debug(.)|if (.==2) then error(.|@j
["DEBUG:",1]
["DEBUG:",2]
jq: error (at <unknown>): 2
```

Debugging

⇒ Use version ≥1.7 where error messages are much improved
or `gojq` as alternative

⇒ Use `stderr` or `debug`for debug output!

⇒ Use`error/1` which will exit with an error message

⇒ Use `halt_error/1` which will exit with specified error code

⇒ Use `if` to check for certain conditions

⇒ Use `try` `catch` around blocks which you think are erroneous

⇒ Use `empty`to end pipelines before errors happen
(especially if you get `jq: error (at <unknown>): `)

⇒ Use a JQ REPL (no good ones out there)
`fq` can do with `fq -i` but doesn't support vars!)

Common Mistakes

- ⇒ Use right number of arguments on `undefined function`: ?
- ⇒ Use files for correct line numbers!
- ⇒ Use parenthesis on *unexpected token*, *syntax* or otherwise *strange error*

```
$ jq -rnc '"foo"|try .+="bar" catch "moo"'
jq: error: syntax error, unexpected catch, expecting
jq: end of file (Unix shell quoting issues?) at <top-le
jq: 1 compile error

# Now the parser will understand it correctly:
$ jq -rnc '"foo"|try (.+="bar") catch "moo"'
foobar
```

Own Debug Functions

If `debug` output `["DEBUG:", <current-value>]` doesn't suite you *just* overwrite!

```
def debug:
    \(if (type=="string") then  (@text|@json) else @jso

def debug(msgs):
    (msgs | debug | empty), .;
```

IDE Plugins

⇒ For a convenient jq development experience:

- ⇒ jq-dash-docset
- ⇒ vscode-jq
- ⇒ jq-lsp

Developing JQ (from JQJQ)

- ⇒ `jq -n --debug-dump-disasm '...'` show jq byte code
- ⇒ `jq -n --debug-trace=all '...'` show jq byte code run trace
- ⇒ `jq -n '{a: "hello"} | debug' 2> >(jq -R 'gsub("\u001b\\[.*?m";"") | fromjson' >&2)` pretty print debug messages
- ⇒ `GOJQ_DEBUG=1 go run -tags gojq_debug cmd/gojq/main.go -n '...'` run gojq in debug mode
- ⇒ `fq -n '".a.b" | _query_fromstring'` gojq parse tree for string
- ⇒ `fq -n '{...} | _query_tostring'` jq expression string for gojq parse tree

Notable JQ Implementations and Wrappers

There are many alternative implementations, library and CLI wrappers adding format support for YAML.

But just some of them are *good*.

Check Official Alternative jq implementations

💙 `gojq` – *adopt!*

⇒ Re-Implementation in pure golang,

⇒ better error messages and

⇒ correct implementation based on Denotational Semantics and a Fast Interpreter for jq (https://arxiv.org/abs/2302.10576)

⇒ basically complete (2.8K stars, very few minor bugs)

⇒ pros and cons are discussed at jq-wiki: Regarding gojq

💙 `fq` – *just as good as gojq and better!*

⇒ Based on `gojq` and understands itself as `jq` for a ton of binary formats (images, audio, archives, ...)

⇒ Example: List files in a Docker Image

```
docker save repo/image \
   | fq -r '
           [(.files[] | select(.name=="manifest.json").
           |[.files[] | select(.name==$l).data.files[].
           |add[]
           |tostring
'
```

⇒ Serialization formats: JSON, BSON, Bencode, YAML, XML, ASN1 BER, Avro, CBOR, protobuf, ... *(!)*

💙 `oq` *– activley maintained*

⇒ It's the best wrapper for original jq (also listed in `jq` wiki)

⇒ It handles YAML and XML – *but not TOML (since it is written in Crystal!)*

⇒ It translates formats and retains meta ass much as possible (including YAML v1.2 with anchors)

⇒ Respects `jq` CLI options correctly

Supplementary Tooling

Closing Thoughts

JQ as Query Language and joining JSONs

Let's `LEFT JOIN` two JSON files we got from an API.

```
jq -cnr \
  --slurpfile instances instances.json \
  --slurpfile images images.json '
  $instances[][]
  |.Image=(
    (
      .ImageId as $imageId
      |$images[][]
      |select(.Id==$imageId)
    )
    // null
  )'
```

```
{ "Id": 5, "ImageId": 1, "Message": "foo", "Image": { "
{ "Id": 5, "ImageId": 4, "Message": "foo", "Image": nul
…
```

You can. But just don't! It can easily get more complex.

Refactor

```
jq -nr \
  --slurpfile instances instances.json \
  --slurpfile images images.json '
  def get_image($image_id):
    (.ImageId as $imageId|$images[][]|select(.Id==$imag

  $instances[][]|.Image=get_image(.ImageId)
'
```

I do think pure SQL is easier!

I do think SQL is easier to understand:

```
SELECT ins.*, img.*
FROM `instances.json` ins
LEFT JOIN `images.json` img  ON ins.ImageId=img.Id
```

Next time: Infrastructure as SQL

With CoudQuery and Steampipe.

Resources

General

⇒ JQ Homepage | Download JQ

⇒ JQ Manual | JQ Play | JQ Cookbook | JQ Wiki

⇒ Github: Awesome list of JQ

⇒ Github: Awesome list of JSON

⇒ Make sure you read the FAQ!

Libraries and Recipes

- ⇒ JBOL is a collection of modules and tools for the **JQ** language.
- ⇒ Rosetta Code on JQ
- ⇒ Offical JQ Cookbook
- ⇒ My ~/.jq dotfile
  - ⇒ GRON
  - ⇒ JSON Objects to Tables / Markdown / CSV
  - ⇒ AWS Config / INI Reader
  - ⇒ AWS, K8s helpers
  - ⇒ OTher useful functions
- ⇒ My Ansible JQ Filter Plugin

Selected Articles

⇒ Golang Implementation of `jq` (`gojq`) *– an author of goqj speaks here*

⇒ Thoughtbot.com: JQ is sed fot JSON

⇒ Medium.com Search for JQ

⇒ Wikipedia on JQ

⇒ jq-wiki: Language Description

JQ Implementations

⇒ JQ in JavaScript

⇒ Jackson JQ pure Java JQ implementation

⇒ Java Wrapper for JQ

JMESPath

- ⇒ JMESPath Spec – *now how far is a spec away from a manual?*
- ⇒ JMESPath Implementations and libraries
- ⇒ JMESPath CLI – *yes, you also can use JMESPath outside of the AWS CLI (uses JMESPath in Golang library)*
- ⇒ REDIS JMESPath Custom Functions – *which were highly needed for Redis*
- ⇒ jc is Python Project to allow JSON output for Linux/GNU/** tools that yet do not support it-
- ⇒ Article: Parsing Command Output in Ansible with JC

Selected Tools

- ⇒ gron – *JSON Javascript line-by-line notation*
- ⇒ GRON implemented in AWK
- ⇒ JSON dot path in AWK
- ⇒ JSONPath.sh Pure JSONPath implementation in Bash for filtering, merging and modifying JSON

Other JSON Query Languages and Tools *(!)*

⇒ dasel
– a new and slick tool for simple editing for every config file (TOML, INI, JSON, YAML)
– *looks pretty promising for DevOps tooling!*

⇒ JSONiq – *The SQL of NoSQL / inspired by XQuery*

⇒ JSONPath IETF draft
– *from the people that cursed the world with java-based XML/XSLT processors*
– *yes, there is an active project for JSONPath for Java*!

⇒ JSONata – *also check JQ vs. JSONata*

⇒ JSLT - *JSON query and transformation language. The language design is inspired by jq, XPath, and XQuery.*

⇒ JSONNet and JSONPatch – *shall be known from Kubernetes*

⇒ JSONPath Online Evaluater

Final Thoughts

Questions?