

# Relatório

Gabriel Inácio

[gabriel.inacio@telcomanager.com](mailto:gabriel.inacio@telcomanager.com)

[github.com/GabrielIDSM/JSON-And-Queries](https://github.com/GabrielIDSM/JSON-And-Queries)

12/2020

# Sumário

<b>1</b>	<b>C/C++</b>	<b>3</b>
1.1	jq - Command-line JSON processor . . . . .	3
1.1.1	Introdução . . . . .	3
1.1.2	Sintaxe . . . . .	3
1.1.3	Filtros Básicos . . . . .	3
1.1.4	Consulta Simples . . . . .	4
1.1.5	Where . . . . .	5
1.1.6	Group By . . . . .	5
1.1.7	Join . . . . .	6
1.1.8	Funções de Agregação . . . . .	6
<b>2</b>	<b>JavaScript</b>	<b>8</b>
2.1	JSONata - Query and transformation language . . . . .	8
2.1.1	Introdução . . . . .	8
2.1.2	Linguagem de Query e Exemplos . . . . .	8
2.2	fx - Command-line tool and terminal JSON viewer . . . . .	9
2.2.1	Introdução . . . . .	9
2.2.2	Sintaxe . . . . .	9
2.2.3	Consulta Simples . . . . .	9
2.2.4	Where . . . . .	9
2.2.5	Group By . . . . .	9
2.2.6	Join . . . . .	10
2.2.7	Funções de Agregação . . . . .	10
2.3	emuto - JSON Processor . . . . .	11
2.3.1	Introdução . . . . .	11
2.3.2	Sintaxe . . . . .	11
2.3.3	Scripts . . . . .	11
2.3.4	Linguagem de Query . . . . .	11
2.3.5	Consulta Simples . . . . .	11
2.3.6	Funções de Agregação . . . . .	12
2.4	json-query - Retrieves values from JSON objects for data binding . . . . .	13
2.4.1	Introdução . . . . .	13
2.4.2	Linguagem de Query e Exemplos . . . . .	13
<b>3</b>	<b>Python</b>	<b>14</b>
3.1	jello - Filter JSON and JSON Lines data . . . . .	14
3.1.1	Introdução . . . . .	14
3.1.2	Sintaxe . . . . .	14
3.1.3	Consulta . . . . .	14
<b>4</b>	<b>Lua</b>	<b>16</b>
4.1	JMESPath.lua . . . . .	16
4.1.1	Introdução . . . . .	16
4.1.2	Linguagem de Query e Exemplos . . . . .	16

# 1 C/C++

## 1.1 jq - Command-line JSON processor

### 1.1.1 Introdução

O projeto jq o define como “*Um processador de linha de comando para arquivos JSON leve e flexível*”. Ele foi desenvolvido com o intuito de ser como o *sed* para arquivos JSON.

É um projeto de código aberto, com código disponível no GitHub (<https://github.com/stedolan/jq>).

### 1.1.2 Sintaxe

Considerando um contexto ideal de uma biblioteca que recebe um JSON com uma query e retorna um JSON, o jq consegue realizar isso por meio de **filtros**. Os filtros não são tão avançados quanto uma query SQL, porém conseguem entregar um conjunto de informações restrita que pode ser utilizada.

### 1.1.3 Filtros Básicos

#### Identidade

```
$ jq '.'
```

#### Identificador de Objeto-Índice

```
$ jq '.foo'
```

```
Input    {"foo": 42, "bar": "less_interesting_data"}
```

```
Output   42
```

```
$ jq '.foo'
```

```
Input    {"notfoo": true, "alsonotfoo": false}
```

```
Output   null
```

#### Identificador de Objeto-Índice Opcional

```
$ jq '.foo?'
```

```
Input    {"foo": 42, "bar": "less_interesting_data"}
```

```
Output   42
```

```
$ jq '.foo?'
```

```
Input    {"notfoo": true, "alsonotfoo": false}
```

```
Output   null
```

#### Arrays

```
$ jq '.[0]'
```

```
Input    [{"name": "JSON", "good": true}, {"name": "XML", "good": false}]
```

```
Output   {"name": "JSON", "good": true}
```

```
$ jq '.[2]'
```

```
Input    [{"name": "JSON", "good": true}, {"name": "XML", "good": false}]
```

```
Output   null
```

## Slice

```
$ jq '.[2:4] '  
Input    ["a","b","c","d","e"]  
Output   ["c","d"]
```

```
$ jq '.[2:4] '  
Input    "abcdefghi"  
Output   "cd"
```

```
$ jq '[:3] '  
Input    ["a","b","c","d","e"]  
Output   ["a","b","c"]
```

## Vírgula

```
$ jq '.foo,_.bar '  
Input    {"foo": 42, "bar": "something_else", "baz": true}  
Output   42  
         "something_else"
```

```
$ jq '.user,_.projects[] '  
Input    {"user":"stedolan", "projects": ["jq", "wikiflow"]}  
Output   "stedolan"  
         "jq"  
         "wikiflow"
```

## Barra Vertical (Pipe)

```
jq '.[]|_.name '  
Input    [{"name":"JSON", "good":true}, {"name":"XML", "good":false}]  
Output   "JSON"  
         "XML"
```

A documentação está disponível em <https://stedolan.github.io/jq/manual>.

### 1.1.4 Consulta Simples

Considerando o mesmo arquivo “Request.json”, podemos obter, por exemplo, o atributo “params” de cada um dos objetos presentes no array “report”:

```
$ jq '.reports|.[]|.params' Request.json
```

O resultado desse comando será:

```
1 {  
2   "params": 0,  
3   "params": 6  
4 }
```

O filtro “`.[]`” tem a função de percorrer todos os elementos do array “reports”.

### 1.1.5 Where

O jq possui uma cláusula equivalente ao Where do SQL, a cláusula **select(bool)**. Como exemplo, podemos obter o objeto no array “reports” que possui o atributo “path” igual a “cli.js”:

```
$ jq '.reports | .[] | select (.path=="cli.js")' Request.json
```

O resultado desse comando será:

```
1 {
2   "aggregate": {
3     "sloc": {
4       "logical": 5,
5       "physical": 7
6     },
7     [...],
8     "path": "cli.js"
9   }
```

### 1.1.6 Group By

É possível também utilizar a cláusula **Group By** nas consultas. Por exemplo, podemos organizar os objetos presentes no array “reports” em função do atributo “loc”:

```
$ jq '.reports | group-by (.loc)' Request.json
```

O resultado desse comando será:

```
1 {
2   [
3     [
4       {
5         [...]
6         "loc": 3.5,
7         [...]
8       }
9     ],
10    [
11      {
12        [...]
13        "loc": 5,
14        [...]
15      }
16    ]
17  ]
18 }
```

### 1.1.7 Join

Existe também suporte para operações de **Join**. O jq possui operadores *SQL-Style*, que são o **IN**, **INDEX** e **JOIN**.

Parecido com tabelas SQL, o Join do jq possui a capacidade de criar um novo JSON a partir de outros 2 arquivos. Porém, o jq possui **suas próprias operações** para isso. Como exemplo, podemos utilizar dois arquivos JSON:

```
1 {
2   [
3     { "name": "User1", "registration": "2020-04-18" },
4     { "name": "User2", "registration": "2020-11-17" }
5   ]
6 }
```

```
1 {
2   [
3     { "name": "User1", "editcount": 164 },
4     { "name": "User2", "editcount": 150 },
5     { "name": "User3", "editcount": 10 }
6   ]
7 }
```

Para obtermos um único arquivo, podemos utilizar o seguinte comando:

```
$ jq -s '[.[0]+.[1]|group-by(.name)|select(length>1)|add]'
1.json 2.json
```

O resultado será:

```
1 {
2   [
3     {
4       "name": "User1",
5       "registration": "2020-04-18",
6       "editcount": 164
7     },
8     {
9       "name": "User2",
10      "registration": "2020-11-17",
11      "editcount": 150
12    }
13  ]
14 }
```

A cláusula Join não foi utilizada, porém foi obtido o resultado esperado.

### 1.1.8 Funções de Agregação

Por padrão, o jq não possui funções de agregação, diferente de bancos de dados SQL. Porém, o jq possui **suporte a funções e procedimentos**, de forma que funções de agregação (como SUM e COUNT de SQL) possam ser definidas para realizar consultas.

Por exemplo, vamos considerar o seguinte JSON:

```

1 {
2   {
3     "owner_id": 1,
4     "owner": "Adams",
5     "age": 25,
6     "pet_id": 10,
7     "pet": "Bella",
8     "litter": 4
9   },
10  { "owner_id": 1,
11    "owner": "Adams",
12    "age": 25,
13    "pet_id": 20,
14    "pet": "Lucy",
15    "litter": 2
16  }
17 }

```

Podemos obter o número de animais por dono, para isso é possível usar o código:

```

$ jq -sc 'def_count($k):_group_by(.[ $k ])[0]|length_as_$l|. [0]|
.pets_count=$l|del(. pet_id , . pet , . litter );
count("owner_id")' 1.json

```

O resultado será:

```

1 {
2   {
3     "owner_id": 1,
4     "owner": "Adams",
5     "age": 25,
6     "pets_count": 2
7   }
8 }

```

## 2 JavaScript

### 2.1 JSONata - Query and transformation language

#### 2.1.1 Introdução

JSONata é um projeto extremamente consolidado para consultas e transformação de dados em JSON. Sua versão mais recente foi publicada em 26 de outubro de 2020, a *1.8.4*.

#### 2.1.2 Linguagem de Query e Exemplos

Como exemplo, é possível acessar o projeto disponível na seguinte página: [https://github.com/GabrielIDSM/JSON-And-Queries/tree/master/Projects/JSONata Project](https://github.com/GabrielIDSM/JSON-And-Queries/tree/master/Projects/JSONata%20Project).

```
QUERY 3.1 (GROUP BY): reports[0].dependencies{type : line}
RESULT:

{ CommonJS: [ 2, 3, 4, sequence: true ] }

TYPEOF: object

QUERY 3.2 (GROUP BY): reports[0].dependencies{type : {'line' : line, 'path' : path}}
RESULT:

{
  CommonJS: {
    line: [ 2, 3, 4, sequence: true ],
    path: [
      './estimator',
      './cumulative_flow_diagram.json',
      './throughput.json',
      sequence: true
    ]
  }
}

TYPEOF: object

QUERY 4.1 (AGG FUNCTIONS): {'values' : $sum(reports.params)}
RESULT:

{ values: 1.5 }

TYPEOF: object

QUERY 4.2 (AGG FUNCTIONS): {'values' : $max(reports.params)}
RESULT:

{ values: 1.5 }

TYPEOF: object

QUERY 4.3 (AGG FUNCTIONS): {'values' : $min(reports.params)}
RESULT:

{ values: 0 }

TYPEOF: object
```



## 2.2 fx - Command-line tool and terminal JSON viewer

### 2.2.1 Introdução

O fx foi desenvolvido para ser semelhante ao jq, porém com algumas interessantes diferenças. O fx possui suporte para a linguagem **JSONata**, que é uma linguagem de query para arquivos JSON, além disso é possível usar **JavaScript** para realizar consultas.

O fx possui dois modos de uso: **CLI** e **Interativo**. O modo interativo funciona como o **VIM**, enquanto o modo CLI é usado como o jq.

### 2.2.2 Sintaxe

Considerando o uso da ferramenta, o modo mais relevante é o **CLI com JSONata**. Dessa forma, basta conhecer a linguagem de query para realizar consultas.

### 2.2.3 Consulta Simples

Para obter o atributo “maintainability” do primeiro objeto no array “reports”, é possível usar:

```
$ fx Request.json 'jsonata("reports[0].maintainability")'
```

### 2.2.4 Where

Para obtermos o objeto “reports” que tem o atributo “params” equivalente a “0”, podemos utilizar:

```
$ fx Request.json 'jsonata("reports[params=0]")'
```

O resultado será:

```
1 {
2   "aggregate": {
3     "sloc": {
4       "logical": 5,
5       "physical": 7
6     },
7     [...],
8     "params": 0,
9     "path": "cli.js"
10  }
```

### 2.2.5 Group By

Para obtermos todos os atributos “path” do array “dependencies” do primeiro objeto do array “reports” organizados em um array chamado “Caminho”, podemos fazer:

```
$ fx Request.json 'jsonata("reports[0].dependencies{ 'Caminho' : _path })'
```

O resultado será:

```
1 {
2   "Caminho": [
3     "./estimator",
4     "./cumulative_flow_diagram.json",
5     "./throughput.json"
```

```
6 ]
7 }
```

### 2.2.6 Join

Por trabalhar com um único arquivo JSON, o JSONata não possui suporte para a cláusula Join do SQL (Apesar de ser possível simular).

### 2.2.7 Funções de Agregação

JSONata possui suporte nativo para funções de agregação, assim é muito simples utilizar. Por exemplo, podemos obter a soma do atributo “params” de todos os objetos do array “reports”:

```
$ fx Request.json 'jsonata("{} values":_:$sum(reports.params))'
```

O resultado será:

```
1 {
2   "values": 1.5
3 }
```

Para que o resultado fosse um objeto JSON, foi necessário usar um **Group By**.

## 2.3 emuto - JSON Processor

### 2.3.1 Introdução

O emuto é semelhante ao jx em questão de funcionamento e sintaxe. Pode ser instalado pelo npm usando:

```
$ npm install -g emuto emuto-cli
```

É um projeto de código aberto e possui uma documentação muito bem escrita. É um projeto relativamente menor que o jx, porém igualmente confiável.

Possui um simulador online presente em [https://kantord.github.io/emuto/docs/try\\_emuto](https://kantord.github.io/emuto/docs/try_emuto).

### 2.3.2 Sintaxe

O emuto pode ser utilizado da seguinte forma:

```
$ cat file.json | emuto 'query'
```

Sua linguagem de query é baseada em **GraphQL**, assim são consideravelmente semelhantes.

O emuto não foi projetado para ser uma linguagem exclusivamente de query, e sim uma linguagem de manipulação e processamento de arquivos JSON, em função disso sua sintaxe não é idêntica a GraphQL.

### 2.3.3 Scripts

Um diferencial do emuto para outras ferramentas é a possibilidade de criar um script para realizar a consulta em um JSON. Dessa forma, basta criar um arquivo com extensão *.emu* e usar da seguinte forma:

```
$ cat file.json | ./file.emu
```

Como exemplo de um script *.emu* temos:

```
#!/ emuto -s

$
| map ($ => $ { commit { message } committer { login } } )
| map ($ => {
    "committer": $.committer.login ,
    "message":   $.commit.message ,
  })
```

### 2.3.4 Linguagem de Query

Sua linguagem de query tem o objetivo de criar uma nova estrutura a partir de um JSON existente. Dessa forma, pode-se dizer que o usuário cria a estrutura final do JSON enquanto o emuto preenche os campos.

### 2.3.5 Consulta Simples

Para obter o atributo "loc", basta realizar a seguinte query:

```
$ cat Request.json | emuto '$.loc'
```

### 2.3.6 Funções de Agregação

Para obtermos, por exemplo, o número de objetos no array “reports”, basta executar-mos a query:

```
$ cat Request.json | emuto '$_{...Stats_}
```

where

```
$_$Stats=($=>{  
    $_$_"Reports_Length":($_.reports|length)  
    $_$_}) '
```

O resultado será:

```
1 {  
2   "Reports_Length": 2  
3 }
```



## 3 Python

### 3.1 jello - Filter JSON and JSON Lines data

#### 3.1.1 Introdução

O jello foi desenvolvido para ser, resumidamente, “*um jq que utiliza uma query em python*”. Dessa forma, ele possui as mesmas funcionalidades do jq, porém utiliza a sintaxe de Python para realizar consultas.

Essa ferramenta é relativamente recente, a mais nova de todas as ferramentas citadas. Em função disso, não possui muitos contribuidores. Ela é desenvolvida por **Kelly Brazil** e é disponibilizada sob a licença do MIT.

Seu código pode ser encontrado no GitHub na página <https://github.com/kellyjonbrazil/jello>.

Sua instalação pode ser feita via terminal:

```
$ pip3 install jello
```

Para usar o *pip* basta ter o Python instalado.

#### 3.1.2 Sintaxe

Para executar uma query em um arquivo JSON usando o jello, basta seguir o seguinte padrão:

```
$ cat file.json | jello 'op' 'query'
```

Sua query utiliza da sintaxe pura de **Python 3**, assim basta conhecer a linguagem para conseguir construir uma query.

Uma *feature* muito útil do jello é a implementação do módulo **glom**. O glom é um módulo de manipulação de dados para Python, útil para operações em estruturas aninhadas.

Para poder usar o glom, é necessário adicionar a seguinte linha no arquivo *.jelloconf.py*:

```
from glom import *
```

#### 3.1.3 Consulta

Apesar de ser uma ferramenta prática, sua linguagem de query possui muitas limitações. Por ser baseada em Python, consultas complexas exigem grande esforço por parte do programador para desenvolver a query.

Por exemplo, para listar todos os objetos no array “reports” devemos fazer:

```
$ cat Request.json | jello '\nresult=[\nfor _data_in_ _["reports"]:\n_ _result.append(data)\nresult'
```

O resultado da query será:

```
1 {
2   [
3     {
4       "aggregate": {
5         "sloc": {
6           "logical": 5,
7           "physical": 7
8         },
9         [...],
10        "path": "cli.js"
11      },
12      {
13        "aggregate": {
14          "sloc": {
15            "logical": 20,
16            "physical": 62
17          },
18          [...],
19          "path": "estimator.js"
20        }
21      ]
22    }
```

## 4 Lua

### 4.1 JMESPath.lua

#### 4.1.1 Introdução

JMESPath é um projeto desenvolvido por James Saryerwinnie disponibilizado em linguagens como Lua, JavaScript, PHP, Python, Java, Ruby, Go e etc. Para fins práticos, os testes serão feitos utilizando a biblioteca **jmespath.lua**.

Essa biblioteca tem o objetivo de executar queries em arquivos JSON e obter um novo JSON como resposta. Sua linguagem de query é extremamente complexa e possui suporte para diversos operadores.

Infelizmente, a última atualização que a ferramenta recebeu foi em 2014. Além disso, não existe suporte para operações de Join e Group By.

#### 4.1.2 Linguagem de Query e Exemplos

Como exemplo, é possível acessar o projeto disponível na seguinte página: ***[https://github.com/GabrielIDSM/JSON-And-Queries/tree/master/Projects/JMESPath Project](https://github.com/GabrielIDSM/JSON-And-Queries/tree/master/Projects/JMESPath%20Project)***.