Relatório

Gabriel Inácio
gabriel.inacio@telcomanager.com
github.com/GabrielIDSM/JSON-And-Queries

12/2020

Sumário

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:

```
{
  "params": 0,
  "params": 6
}
```

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:

```
{
  "aggregate": {
    "sloc": {
      "logical": 5,
      "physical": 7
    },
  [...],
  "path": "cli.js"
}
```

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:

```
{
[
  [
    {
      [...]
      "loc": 3.5,
      [...]
    }
  ],
  [
    {
      [...]
      "loc": 5,
      [...]
    }
  ]
]
}
```

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:

```
{
  [
    { "name": "User1", "registration": "2020-04-18" },
    { "name": "User2", "registration": "2020-11-17" }
  ]
}
```

```
{
  [
    { "name": "User1", "editcount": 164 },
    { "name": "User2", "editcount": 150 },
    { "name": "User3", "editcount": 10 }
  ]
}
```

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:

```
{
  [
    {
      "name": "User1",
      "registration": "2020-04-18",
      "editcount": 164
    },
    {
      "name": "User2",
      "registration": "2020-11-17",
      "editcount": 150
    }
  ]
}
```

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:

```
{
  {
    "owner_id": 1,
    "owner": "Adams",
    "age": 25,
    "pet_id": 10,
    "pet": "Bella",
    "litter": 4
  },
  { "owner_id": 1,
    "owner": "Adams",
    "age": 25,
    "pet_id": 20,
    "pet": "Lucy",
    "litter": 2
  }
}
```

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

```
$ jq -sc 'def count($k): group_by(.[$k])[]|length as $l|.[0]|
.pets_count=$l|del(.pet_id,.pet,.litter);
count("owner_id")' 1.json
```

O resultado será:

```
{
  {
    "owner_id": 1,
    "owner": "Adams",
    "age": 25,
    "pets_count": 2
  }
}
```

2 JavaScript

2.1 fx - Command-line tool and terminal JSON viewer

2.1.1 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.

2.1.2 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.

2.1.3 Consulta Simples

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

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

2.1.4 Where

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

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

O reseultado será:

```
{
  "aggregate": {
    "sloc": {
      "logical": 5,
      "physical": 7
    },
  [...],
  "params": 0,
  "path": "cli.js"
}
```

2.1.5 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:

```
{
  "Caminho": [
    "./estimator",
    "./cumulative_flow_diagram.json",
    "./throughput.json"
  ]
}
```

2.1.6 Join

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

2.1.7 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:

```
{
  "values": 1.5
}
```

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

2.2 emuto - JSON Processor

2.2.1 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***.

2.2.2 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.

2.2.3 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,
    })
```

2.2.4 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.

2.2.5 Consulta Simples

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

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

2.2.6 Funções de Agregação

Para obtermos, por exemplo, o número de objetos no array "reports", basta executarmos a query:

```
$ cat Request.json | emuto '$ { ... Stats }

where
  $Stats=($=>{
      "Reports Length":($.reports|length)
    })'
```

O resultado será:

```
{
  "Reports Length": 2
}
```

2.3 json-query - Retrieves values from JSON objects for data binding

2.3.1 Introdução

O json-query é uma biblioteca JavaScript desenvolvida para extrair informações de JSON e usá-los de maneira nativa. Infelizmente, o projeto não recebe modificações desde 2017. Sua sintaxe é bastante simples e tem o objetivo de lidar mais com o objeto nativo do que com o resultado da query. Dessa forma, o json-query precisa de uma grande quantidade de código pré-definido para obter resultados complexos.

2.4 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/json-query Project***.

```
==========JSON-QUERY PROJECT==========


QUERY 2 (WHERE): reports[params=0].maintainability
RESULT:

124.63257712933509

TYPEOF: number

QUERY 1 (BASIC): reports
RESULT:

[
  {
    aggregate: {
      sloc: [Object],
      cyclomatic: 1,
      halstead: [Object],
      params: 0,
      line: 2,
      cyclomaticDensity: 20
    },
    functions: [],
    dependencies: [ [Object], [Object], [Object] ],
    maintainability: 124.63257712933509,
    loc: 5,
    cyclomatic: 1,
    effort: 377.68960826811434,
    params: 0,
    path: 'cli.js'
  },
```

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 '\
result=[]
for␣data␣in␣␣["reports"]:
␣␣result.append(data)
result'
```

O resultado da query será:

```
{
  [
    {
      "aggregate": {
      "sloc": {
        "logical": 5,
        "physical": 7
      },
      [...],
      "path": "cli.js"
    },
    {
      "aggregate": {
      "sloc": {
        "logical": 20,
        "physical": 62
      },
      [...],
      "path": "estimator.js"
    }
  ]
}
```

4 Lua

4.1 JMESPath

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 feito 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.

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***.