Introduction to jq - part 2 (HPR Show 4114)

Options to jq; learning about filters

Dave Morriss

Overview

In the last episode we looked at how JSON data is structured and saw how `jq` could be used to format and print this type of data.

In this episode we'll visit a few of the options to the `jq` command and then start on the filters written in the `jq` language.

Options used by `jq`

In general the `jq` command is invoked thus:

```
jq [options...] filter [files...]
```

It can be given data in files or sent to it via the STDIN (standard in) channel. We saw data being sent this way in the last episode, having been downloaded by `curl`.

There are many options to the command, and these are listed in the manual page and in the online manual. We will look at a few of them here:

`--help` *or* `-h`

> Output the `jq` help and exit with zero.

`-f filename` *or* `--from-file filename`

> Read filter from the file rather than from a command line, like awk´s -f option. You can also use ´#´ to make comments in the file.

`--compact-output` *or* `-c`

> By default, `jq` pretty-prints JSON output. Using this option will result in more compact output by instead putting each JSON object on a single line.

`--color-output` *or* `-C` *and* `--monochrome-output` *or* `-M`

> By default, `jq` outputs colored JSON if writing to a terminal. You can force it to produce color even if writing to a pipe or a file using `-C`, and disable color with `-M`.

`--tab`

> Use a tab for each indentation level instead of two spaces.

`--indent n`

> Use the given number of spaces (no more than 7) for indentation.

Notes

- The `-C` option is useful when printing output to the `less` command with the colours that `jq` normally generates. Use this:

  ```
  jq -C '.' file.json | less -R
  ```

  The `-R` option to less allows colour escape sequences to pass through.

- Do not do what I did recently. Accidentally leaving the `-C` option on the command caused `formatted.json` to contain all the escape codes used to colour the output:

  ```
  $ jq -C '.' file.json > formatted.json
  ```

This is why `jq` normally only generates coloured output when writing to the terminal.

Filters in `jq`

As we saw in the last episode JSON can contain arrays and objects. Arrays are enclosed in square brackets and their elements can be any of the data types we saw last time. So, arrays of arrays, arrays of objects, and arrays of both of these are all possible.

Objects contain collections of keyed items where the keys are strings of various types and the values they are associated with can be any of the data types.

JSON Examples

Simple arrays:

```
[1,2,3]
[1,2,3,[4,5,6]]
["Hacker","Public","Radio"]
["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"]
```

Simple object:

```
{ "name": "Hacker Public Radio", "type": "podcast"}
```

This more complex object was generated by the Random User Generator API. It is a subset of what can be obtained from this site.

```
{
  "gender": "female",
  "name": {
    "title": "Mrs",
    "first": "Jenny",
    "last": "Silva"
  },
  "dob": {
    "date": "1950-01-03T21:38:19.583Z",
    "age": 74
  },
  "nat": "GB"
}
```

This one comes from the file `countries.json` from the Github project `mledoze/countries`. It is a subset of the entry for *Mexico*.

```
{
  "name": {
    "common": "Mexico",
    "official": "United Mexican States",
    "native": {
      "spa": {
        "official": "Estados Unidos Mexicanos",
        "common": "México"
      }
    }
  },
  "capital": [
    "Mexico City"
  ],
  "borders": [
    "BLZ",
    "GTM",
    "USA"
  ]
}
```

Identity filter

This is the simplest filter which we already encountered in episode 1: `'.'`. It takes its input and produces the same value as output. Since the default action is to pretty-print the output it formats the data:

```
$ echo '["Hacker","Public","Radio"]' | jq .
[
  "Hacker",
  "Public",
  "Radio"
]
```

Notice that the filter is not enclosed in quotes in this example. This is usually fine for the simplest filters which don't contain any characters which are of significance to the shell. It's probably a good idea to always use (single) quotes however.

There may be considerations regarding how `jq` handles numbers. Consult the `jq` documentation for details.

Object Identifier-Index filter

This form of filter refers to object keys. A particular key is usually referenced with a full-stop followed by the name of the key.

In the HPR statistics data there is a top-level key "hosts" which refers to the number of currently registered hosts. This can be obtained thus (assuming the JSON is in the file `stats.json`):

```
$ jq '.hosts' stats.json
357
```

The statistics file contains a key `'stats_generated'` which marks a Unix time value (seconds since the Unix Epoch 1970-01-01). This can be decoded on the command line like this:

```
$ date -d "@$(jq '.stats_generated' stats.json)" +'%F %T'
2024-04-18 15:30:07
```

Here the `'-d'` option to `date` provides the date to print, and if it begins with a `'@'` character it's interpreted as seconds since the Epoch. Note that the result is in my local time zone which is currently UTC + 0100 (aka BST).

Using object keys in this way only works if the keys contain only ASCII characters and underscores and don't start with a digit. To use other characters it's necessary to enclose the key in double quotes or square brackets and double quotes. So, assuming the key we used earlier had been altered to `'stats-generated'` we could use either of these expressions:

```
."stats-generated"
.["stats-generated"]
```

Of course, the `.[<string>]` form is valid in all contexts. Here `<string>` represents a JSON string in double quotes. The `jq` documentation refers to this as an *Object Index*.

What if you want the `next_free` value discussed in the last episode (number of shows until the next free slot)? Just typing the following will not work:

```
$ jq '.next_free' stats.json
null
```

This is showing that there is no key `next_free` at the top level of the object, the key we want is in the object with the key `slot`.

If you request the `slot` key this will happen:

```
$ jq '.slot' stats.json
{
  "next_free": 8,
  "no_media": 0
}
```

Here an object has been returned, but we actually want the value within it, as we know.

This is where we can *chain* filters like this:

```
$ jq '.slot | .next_free' stats.json
8
```

The *pipe* symbol causes the result of the first filter to be passed to the second filter. Note that the pipe here is not the same as the Unix pipe, although it looks the same

There is a shorthand way of doing this "chaining":

```
$ jq '.slot.next_free' stats.json
8
```

This is a bit like a file system path, and makes the extraction of desired data easier to visualise and therefore quite straightforward, I think.

Array index filter

We have seen the object index filter `.[<string>]` where `<string>` represents a key in the object we are working with.

It makes sense for array indexing to be `.[<number>]` where `<number>` represents an integer starting at zero, or a negative integer. The meaning of the negative number is to count backwards from the last element of the array (which is `-1`).

So, some examples might be:

```
$ echo '["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"]' | jq '.[1]'
"Monday"

$ echo '["Sun","Mon","Tue","Wed","Thu","Fri","Sat"]' | jq '.[-1]'
"Sat"

$ echo '[1, 2, 3, [4, 5, 6]]' | jq '.[-1]'
[
  4,
  5,
  6
]
```

We will look at more of the basic filters in the next episode.

Links

- `jq`:
  - GitHub page
  - Downloading `jq`
  - The `jq` manual
  - Wikipedia page about the jq programming language
- Test data sources:
  - Random User Generator API
  - Github project `mledoze/countries`