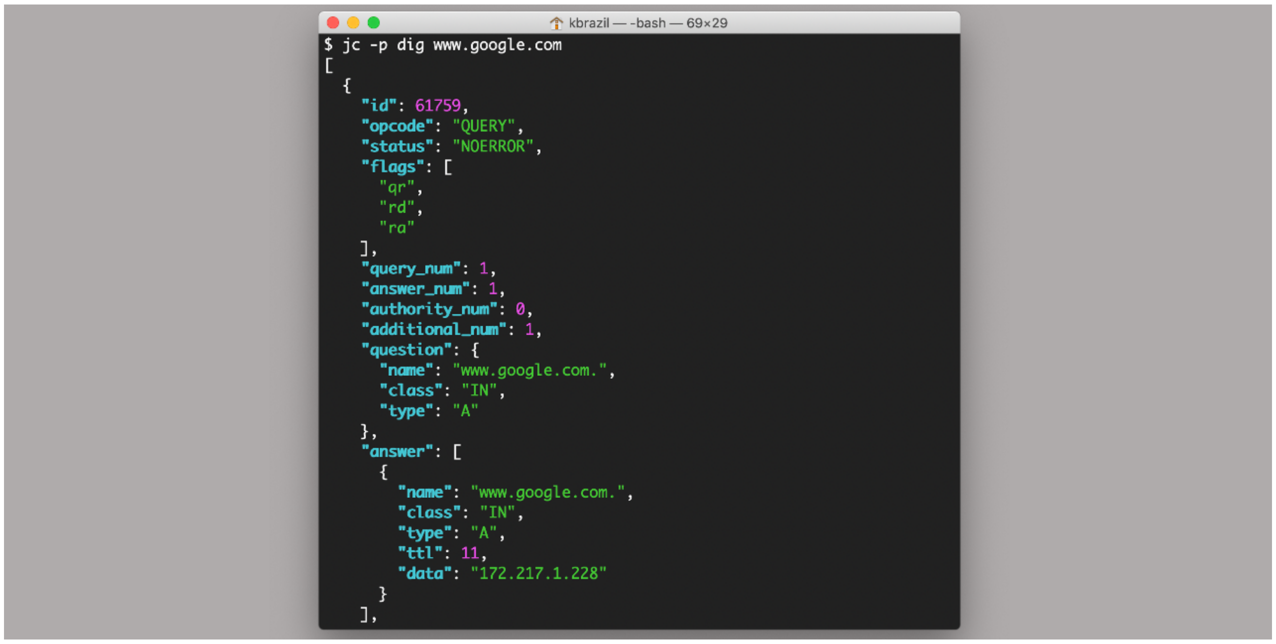
Brazil's Blog

Trivial and profound thoughts on cybersecurity and computing in general.

Tips on Adding JSON Output to Your CLI App

kellyjonbrazil December 3, 2021 All Posts, Programming
best practices, cli, command-line, json, json-lines



```
$ jc -p dig www.google.com
[
  {
    "id": 61759,
    "opcode": "QUERY",
    "status": "NOERROR",
    "flags": [
      "qr",
      "rd",
      "ra"
    ],
    "query_num": 1,
    "answer_num": 1,
    "authority_num": 0,
    "additional_num": 1,
    "question": {
      "name": "www.google.com.",
      "class": "IN",
      "type": "A"
    },
    "answer": [
      {
        "name": "www.google.com.",
        "class": "IN",
        "type": "A",
        "ttl": 11,
        "data": "172.217.1.228"
      }
    ],
```

A couple of years ago I wrote a somewhat controversial article on the topic of Bringing the Unix Philosophy to the 21st Century by adding a JSON output option to CLI tools. This allows easier parsing in scripts by using JSON parsing tools like `jq`, `jello`, `jp`, etc. without arcane `awk`, `sed`, `cut`, `tr`, `reverse`, etc. incantations.

It was controversial because there seem to be a lot of folks who don't think writing bespoke parsers for each task is a big deal. Others think JSON is evil. There are strong feelings as can be seen in response to the article in the comments and also on Hacker News and Reddit.

I'll let the next generation of DevOps practitioners and developers come to their own conclusions on the basis of our arguments, but the tide is already turning. Something that was just wishful thinking a couple years ago is now becoming a reality! Now, more and more command line applications are offering JSON output as an option. And with `jc`, JSON output can even be coaxed out of older command line applications.

Structured Output Support is Increasing

Now, there are many new command line applications that offer structured output as an option, and even some older ones are adding the option. I find that more and more often when a parser is requested for `jc`, if I check the man page for the application, there is already a JSON or XML output option. Some examples include `nvidia-smi`, `ffprobe`, `docker` CLI, and `tree`. Even `ip` now supports JSON output with `ip route`, which wasn't supported when I originally wrote about it in the article.

I recently developed standard and streaming parsers for the `iostat` command and found that versions 11 and above now have a JSON output option. Way to go!

But when looking at the JSON options for some of these commands, I found some things that could be improved.

JSON Output Do's and Don'ts

While developing over 80 parsers for the `jc` project, I stumbled upon some best practices. My first goal was to make getting the data easy when using `jq`, as that was the only CLI JSON processing tool I really used at the time. With that initial goal, and input from scores of users, this is how I try to make the highest quality JSON output:

> **Note:** Many of these are completely subjective and are just my humble opinion. I'm willing to keep an open mind about these choices.

- Do Make a Schema
- Do Flatten the Structure
- Do Output JSON Lines for Streaming Output
- Do Use Predictable Key Names
- Do Pretty Print with Two Spaces or Don't Format at All
- Don't Use Special Characters in Key Names
- Don't Allow Duplicate Keys
- Don't Use Very Large Numbers
- BONUS

Let’s take a look at these in more detail.

Do

Here are some good practices when generating JSON output:

Make a Schema

When possible, which is almost always the case, I document a schema for the JSON output. This allows the user to know where they can always find an attribute and which type they can expect. (string, integer, float, boolean, null, object, array) This also allows you to test the output to make sure it conforms to the schema and there are no bugs.

A schema doesn’t have to be complicated. It can be specified in the documentation, including the man page. I use this simple structure for `jc` documentation:

```
[
  {
    "foo":      string,
    "bar":      float,    # null if blank
    "baz": [
                integer
    ]
  }
]
```

Flatten the Structure

The best case is to output an object or an array of objects (most common) with no further nesting. Sometimes you can prefix an attribute name if nesting is not absolutely necessary. The idea is to make it as easy for the user to grab the value so they don’t need to traverse the data structure to get what they want.

Sometimes this:

```
[
  {
    "cpu": {
      "speed": 5,
      "temp": 33.2
    },
    "ram": {
      "speed": 11,
      "mb": 1024
    }
  }
]
```

Can be turned into this:

```
  [
    {
      "cpu_speed": 5,
      "cpu_temp": 33.2,
      "ram_speed": 11,
      "ram_mb": 1024
    }
  ]
```

This way I can easily filter the data in `jq` or other tools without having to traverse levels. Of course, this is not always possible or desirable, but keeping a flat structure should be considered for user convenience.

This approach is also great for output that contains a long list of items. I'll pick on `iostat` a bit here to make a point – ***but don't take this the wrong way – I'm thrilled that the author of `iostat` has included a JSON output option and in no way want to discount the work put into that***.

The `iostat` JSON output option deeply nests the output like so:

```
  {
    "sysstat": {
      "hosts": [
        {
          "nodename": "ubuntu",
          "sysname": "Linux",
          "release": "5.8.0-53-generic",
          "machine": "x86_64",
          "number-of-cpus": 2,
          "date": "12/03/2021",
          "statistics": [
            {
              "avg-cpu": {
                "user": 0.6,
                "nice": 0.01,
                "system": 1.68,
                "iowait": 0.14,
                "steal": 0,
                "idle": 97.58
              },
              "disk": [
                {
                  "disk_device": "dm-0",
                  "tps": 29.07,
                  "kB_read/s": 502.25,
                  "kB_wrtn/s": 54.94,
                  "kB_dscd/s": 0,
                  "kB_read": 251601,
                  "kB_wrtn": 27524,
                  "kB_dscd": 0
                },
  ...
```

This makes sense and is very logical when you look at the output as an entire JSON document, but when dealing with command output from certain commands like **`iostat`**, **`vmstat`**, **`ping`**, **`ls`**, etc. which can have huge – even unlimited – amounts of output, it might make more sense to build the JSON structure into a format that is more easily consumed by tools like **`jq`** to be used in a pipeline.

With this structure, the whole document needs to be loaded before the JSON is considered valid and searchable, though **`iostat`** output can actually go on indefinitely depending on how the command is run.

I took a different approach with the **`jc iostat`** parser by using a flat array of objects and simply using a **`type`** attribute to denote which type of object it is. This allows very easy filtering in **`jq`** or other tools and *also allows consistency with the streaming parser, which I'll get to in another section*.

Here's the **`jc`** version:

```
[
  {
      "percent_user": 0.31,
      "percent_nice": 0.23,
      "percent_system": 0.48,
      "percent_iowait": 0.04,
      "percent_steal": 0.0,
      "percent_idle": 98.95,
      "type": "cpu"
    },
    {
      "device": "dm-0",
      "tps": 8.16,
      "kb_read_s": 137.26,
      "kb_wrtn_s": 129.0,
      "kb_dscd_s": 0.0,
      "kb_read": 395021,
      "kb_wrtn": 371240,
      "kb_dscd": 0,
      "type": "device"
    },
    {
      "device": "loop0",
      "tps": 0.01,
      "kb_read_s": 0.12,
      "kb_wrtn_s": 0.0,
      "kb_dscd_s": 0.0,
      "kb_read": 344,
      "kb_wrtn": 0,
      "kb_dscd": 0,
      "type": "device"
    },
  ...
  ]
```

You'll notice that `jc` doesn't bother with metadata around the source of the data that generated the output or even the host statistics. This is because including the source just makes the object nesting deeper without adding value, and the header information is available with other tools like `uname` and `date`, though I could add them in a future parser version as an object with its own `type` if users want that data.

Getting to the data in this structure is pretty easy: just loop over the array, filter by `type` (if needed), and pull attributes from the top-level of each object.

Output JSON Lines for Streaming Output

There's another advantage to the array of flat objects structure discussed above, and that's for programs like `iostat` and others that can stream output forever until the user hits `<ctrl-c>`. In this case, it would be difficult to pipe the output to a JSON filter, like `jq`, since the output would not be valid JSON until the program ends.

For these cases, outputting JSON Lines (aka NDJSON) is a good choice.

Unfortunately, this is what the `iostat` output looks like when running it indefinitely:

```
$ iostat 1 -o JSON
{"sysstat": {
  "hosts": [
    {
      "nodename": "ubuntu",
      "sysname": "Linux",
      "release": "5.8.0-53-generic",
      "machine": "x86_64",
      "number-of-cpus": 2,
      "date": "12/03/2021",
      "statistics": [
        {
          "avg-cpu":  {"user": 1.23, "nice": 0.86, "system":
          "disk": [
            {"disk_device": "dm-0", "tps": 30.16, "kB_read/s"
            {"disk_device": "sr0", "tps": 0.13, "kB_read/s":
          ]
        },
        {
          "avg-cpu":  {"user": 0.00, "nice": 0.00, "system":
          "disk": [
            {"disk_device": "dm-0", "tps": 0.00, "kB_read/s":
            {"disk_device": "sr0", "tps": 0.00, "kB_read/s":
          ]
        },
        {
          "avg-cpu":  {"user": 0.00, "nice": 0.00, "system":
          "disk": [
            {"disk_device": "dm-0", "tps": 5.00, "kB_read/s":
            {"disk_device": "sr0", "tps": 0.00, "kB_read/s":
```

```
            ]
          }
...
```

This is not easily parsable downstream when used in a pipeline:

```
$ iostat 1 -o JSON | jq
^C     # hangs forever until <ctrl-c> is entered and no JSON
```

The author of **`iostat`** did do a cool thing, though, and correctly wrapped the output in the final end brackets when the **`<ctrl-c>`** sequence is caught. So it does finally create a valid JSON document, but I'm not sure all developers will have the forethought to do this. Still, this does not solve the pipelining problem.

Instead, the **streaming `iostat` parser** in **`jc`** outputs JSON lines with the same schema as the **standard parser**. Basically, the only difference is that there are no beginning and ending array brackets and each object is compact printed on its own line. This allows JSON processors like **`jq`** to work on each line immediately as they are emitted:

```
$ iostat 1 | jc --iostat-s -u | jq -c
{"percent_user":1.11,"percent_nice":0.78,"percent_system":1.1
{"device":"dm-0","tps":27.4,"kb_read_s":125.07,"kb_wrtn_s":43
{"device":"loop0","tps":0.02,"kb_read_s":0.16,"kb_wrtn_s":0.0
{"percent_user":2.53,"percent_nice":0.0,"percent_system":1.52
{"device":"dm-0","tps":19.0,"kb_read_s":0.0,"kb_wrtn_s":76.0,
{"device":"loop0","tps":0.0,"kb_read_s":0.0,"kb_wrtn_s":0.0,"
{"percent_user":1.01,"percent_nice":0.0,"percent_system":0.0,
{"device":"dm-0","tps":0.0,"kb_read_s":0.0,"kb_wrtn_s":0.0,"k
{"device":"loop0","tps":0.0,"kb_read_s":0.0,"kb_wrtn_s":0.0,"
...
```

> **Tip:** If you include a JSON Lines output option, you might also want to include an 'unbuffer' option.
>
> When directly printing to the terminal, the OS will disable buffering, but when piping to other programs there will be a buffer typically around 4KB. If the emitted JSON is small, it will look like the terminal is hung. This is why `jc` offers the `-u`, or 'unbuffer' option like many other filtering tools do.
>
> Note, that there may be a performance impact to disabling the buffer, so it should only be disabled while troubleshooting the pipeline in the terminal.

Use Predictable Key Names

This one basically comes down to "don't dynamically generate key names". If key names aren't static and predictable, it makes it difficult to have a good Schema and also makes it difficult for users to find the data.

Instead of doing something like this:

```
{
  "Interface 1": [
    "192.168.1.1",
    "172.16.1.1"
  ],
  "Wifi Interface 1": [
    "10.1.1.1"
  ]
}
```

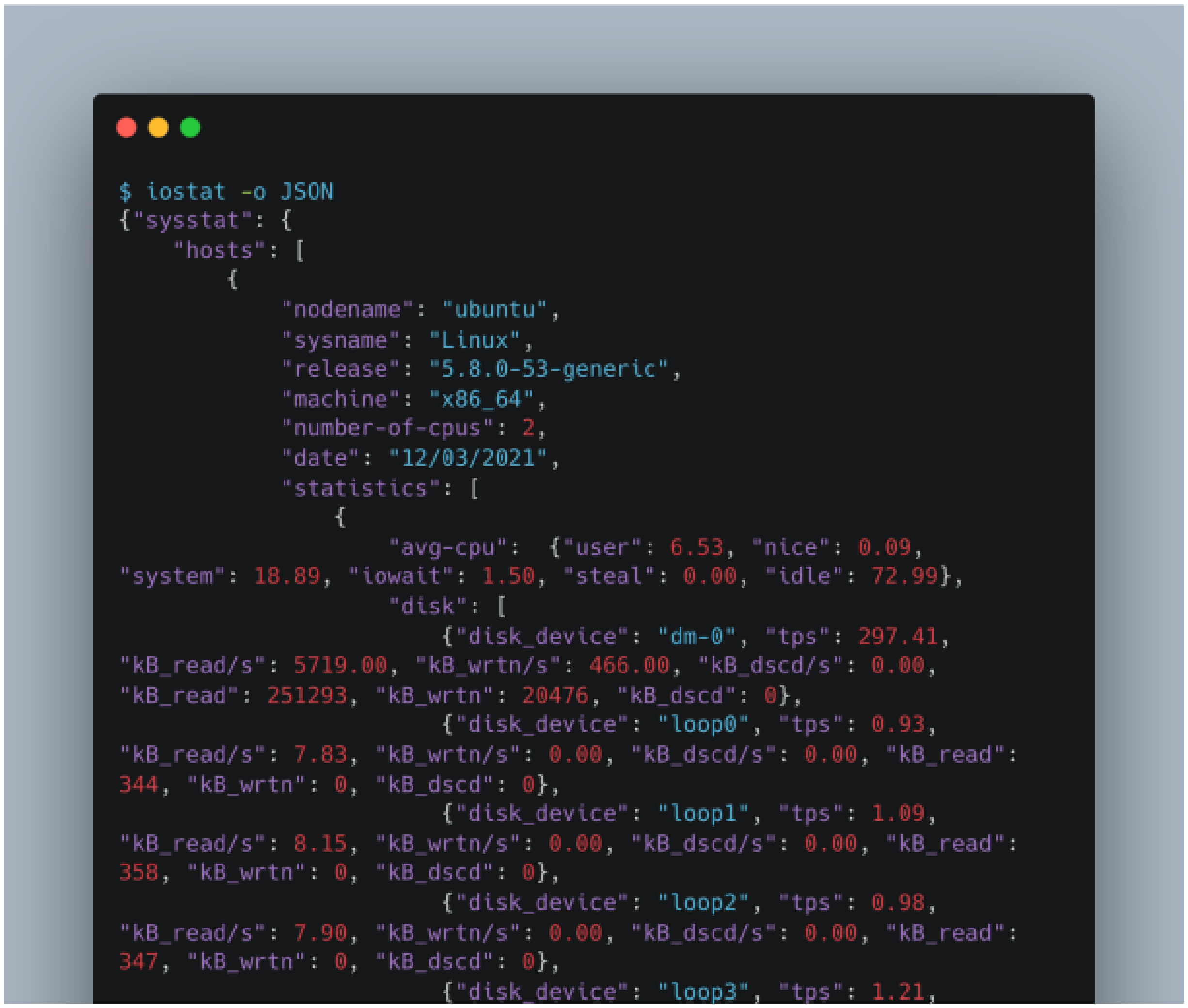
Do this:

```
[
  {
    "interface": "Interface 1",
    "ip_addresses": [
      "192.168.1.1",
      "172.16.1.1"
    ]
  },
  {
    "interface": "Wifi Interface 1",
    "ip_addresses": [
      "10.1.1.1"
    ]
  }
]
```

This is a self-documented structure and the user can simply iterate over all of the objects to get the interface names and IP addresses they want. They can still get it the other way, but it’s not as straightforward and it also doesn’t allow you to have a nicely documented Schema.

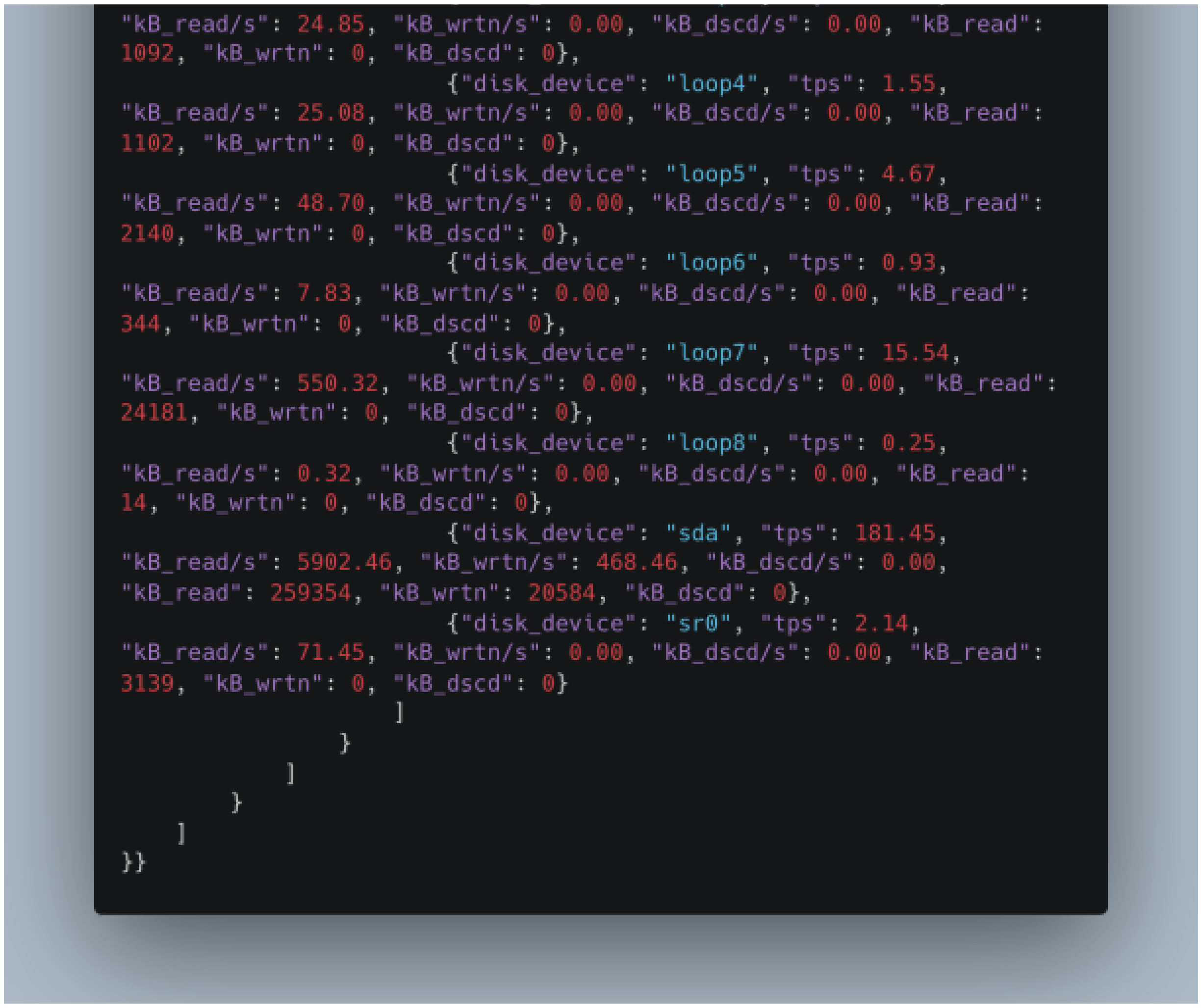
Pretty Print with Two Spaces or Don’t Format at All

Higher-level languages like Python allow very easy formatting of the JSON output, so I typically see the issue of ugly formatted JSON with programs written in C:



```
$ iostat -o JSON
{"sysstat": {
    "hosts": [
        {
            "nodename": "ubuntu",
            "sysname": "Linux",
            "release": "5.8.0-53-generic",
            "machine": "x86_64",
            "number-of-cpus": 2,
            "date": "12/03/2021",
            "statistics": [
                {
                    "avg-cpu":  {"user": 6.53, "nice": 0.09,
"system": 18.89, "iowait": 1.50, "steal": 0.00, "idle": 72.99},
                    "disk": [
                        {"disk_device": "dm-0", "tps": 297.41,
"kB_read/s": 5719.00, "kB_wrtn/s": 466.00, "kB_dscd/s": 0.00,
"kB_read": 251293, "kB_wrtn": 20476, "kB_dscd": 0},
                        {"disk_device": "loop0", "tps": 0.93,
"kB_read/s": 7.83, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
344, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "loop1", "tps": 1.09,
"kB_read/s": 8.15, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
358, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "loop2", "tps": 0.98,
"kB_read/s": 7.90, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
347, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "loop3", "tps": 1.21,
```

]]>



```
"kB_read/s": 24.85, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
1092, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "loop4", "tps": 1.55,
"kB_read/s": 25.08, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
1102, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "loop5", "tps": 4.67,
"kB_read/s": 48.70, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
2140, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "loop6", "tps": 0.93,
"kB_read/s": 7.83, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
344, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "loop7", "tps": 15.54,
"kB_read/s": 550.32, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
24181, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "loop8", "tps": 0.25,
"kB_read/s": 0.32, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
14, "kB_wrtn": 0, "kB_dscd": 0},
                        {"disk_device": "sda", "tps": 181.45,
"kB_read/s": 5902.46, "kB_wrtn/s": 468.46, "kB_dscd/s": 0.00,
"kB_read": 259354, "kB_wrtn": 20584, "kB_dscd": 0},
                        {"disk_device": "sr0", "tps": 2.14,
"kB_read/s": 71.45, "kB_wrtn/s": 0.00, "kB_dscd/s": 0.00, "kB_read":
3139, "kB_wrtn": 0, "kB_dscd": 0}
                    ]
                }
            ]
        }
    ]
}}
```

What is going on here? Actually – I can see what the developer was doing – it does look quite nice outside of the terminal when pasted into a text editor, but while inside the terminal the line wrapping makes it very unreadable.

I like the look of two-space indentation with JSON – maybe because that's the way `jq` formats it and I'm just used to it.

There's really no need to format JSON output at all. If it makes your code simpler, just generate the JSON with no newlines or spaces. This is more compact and the user can just as easily pipe the output through `jq` or other tools to format it.

If you do choose to format the JSON, then take a cue from `jq` and use two spaces of indent and don't coalesce brackets. Like so:

```
$ iostat -o JSON | jq
{
  "sysstat": {
    "hosts": [
      {
        "nodename": "ubuntu",
        "sysname": "Linux",
        "release": "5.8.0-53-generic",
        "machine": "x86_64",
        "number-of-cpus": 2,
        "date": "12/03/2021",
        "statistics": [
          {
            "avg-cpu": {
              "user": 0.6,
              "nice": 0.01,
              "system": 1.68,
              "iowait": 0.14,
              "steal": 0,
              "idle": 97.58
            },
            "disk": [
              {
                "disk_device": "dm-0",
                "tps": 29.07,
                "kB_read/s": 502.25,
                "kB_wrtn/s": 54.94,
                "kB_dscd/s": 0,
                "kB_read": 251601,
                "kB_wrtn": 27524,
                "kB_dscd": 0
              },
              <SNIP>
              {
                "disk_device": "sr0",
                "tps": 0.19,
                "kB_read/s": 6.27,
                "kB_wrtn/s": 0,
                "kB_dscd/s": 0,
                "kB_read": 3139,
                "kB_wrtn": 0,
                "kB_dscd": 0
              }
            ]
            }
          ]
        }
      ]
    }
  }
```

Beggars can't be choosers, so I'll take ugly JSON over no JSON any day. But again, compact JSON with no spaces and newlines is perfectly fine. Anyone working with JSON knows to use `jq` or other tools to make it easy to read in the terminal.

Don't

Try to avoid these JSON smells:

Don't Use Special Characters in Key Names

There's nothing more annoying than having to encapsulate an attribute name in brackets because it has special characters or spaces in it.

```
$ echo '{"Foo/ foo": "bar"}' | jq '.Foo/ foo'
jq: error: foo/0 is not defined at <top-level>, line 1:
.Foo/ foo
jq: 1 compile error

$ echo '{"Foo/ foo": "bar"}' | jq '.["Foo/ foo"]'
"bar"
```

Don't make your users do that! This can also be a consequence of dynamically generating your keys, as discussed in a section above. Instead, keep all key characters lower-case and convert special characters to underscores ('`_`') to keep them alphanumeric.

Underscores are better than dashes because they allow you to select the entire key with a double-click in most IDEs and text editors. Dashes will typically only select a section of the key name.

Don't Allow Duplicate Keys

If you are dynamically generating key names it may be possible for duplicates to appear in an object. If there is a possibility of this, wrap those items in individual objects. Duplicate keys are undefined in JSON and may cause different behavior depending on the client.

Don’t Use Extremely Large Numbers

JSON has nice typing, but unfortunately the numeric data type is underspecified in the standard and may have different behavior with different clients. This can happen if you output a long UUID as a number – the UUID may actually not turn out to be the same on all clients! If you have a very large number, it’s probably best to just wrap it in a string so it doesn’t get mangled downstream.

Don’t Use XML

Just joking! Any standard structured output is better than plain text in many cases, and sometimes (but not often) XML is a better choice than JSON. I prefer JSON for its readability, support ecosystem, and for its support for maps, arrays, and limited types. After developing JSON schemas for over 80 CLI parsers I’ve found that there’s not much JSON can’t do for this type of output.

In Conclusion

Always think of the end-user and how they will interact with the data. By following these steps, you can keep the users from having to jump through extra hoops to get to the data they want:

- Make a Schema
- Flatten the Structure
- Output JSON Lines for Streaming Output
- Use Predictable Key Names
- Pretty Print with Two Spaces or Don’t Format at All
- Don’t Use Special Characters in Key Names
- Don’t Allow Duplicate Keys
- Don’t Use Very Large Numbers

This is clearly not an exhaustive list. Did I miss any of your pet peeves? Let me know in the comments!

Like this:

Like

One blogger likes this.

kellyjonbrazil December 3, 2021 All Posts, Programming
best practices, cli, command-line, json, json-lines

Published by kellyjonbrazil

I'm a cybersecurity and cloud computing nerd. View more posts

Leave a Reply

Enter your comment here...