
fastgron

Make JSON greppable super fast!

fastgron transforms JSON into discrete assignments to make it easier to grep for what you want and see the absolute 'path' to it. It eases the exploration of APIs that return large blobs of JSON but have terrible documentation.

fastgron is a high-performance JSON to GRON converter, developed in C++20, utilizing simdjson library. It's 50x faster than gron on big files (400MB/s input / 1.8GB/s output on M1 Macbook Pro), so it makes big JSON files greppable.

```
1 $ fastgron "https://api.github.com/repos/adamritter/fastgron/commits?
   per_page=1" | fgrep commit.author
2 json[0].commit.author = {}
3 json[0].commit.author.name = "adamritter"
4 json[0].commit.author.email = "58403584+adamritter@users.noreply.github
   .com"
5 json[0].commit.author.date = "2023-05-30T18:04:25Z"
```

fastgron can work backwards too, enabling you to turn your filtered data back into JSON:

```
1 $ fastgron "https://api.github.com/repos/adamritter/fastgron/commits?
   per_page=1" | fgrep commit.author | fastgron --ungron
2 [
3   {
4     "commit": {
5       "author": {
6         "date": "2023-05-30T18:11:03Z",
7         "email": "58403584+adamritter@users.noreply.github.com",
8         "name": "adamritter"
9       }
10    }
11  ]
12 ]
```

Quick Install

- Arch: `yay -S fastgron-git`
- Homebrew: `brew install fastgron --build-from-source`
- Nix: `nix profile install github:adamritter/fastgron#fastgron`
- Ubuntu: Download the latest binary from releases.
- Windows: Download the latest binary from releases. Note that libcurl support is missing from the released binary, so [http](#) and [https](#) URLs can't yet be read directly.

Usage

```
1 $ cat testdata/two.json
2 {
3     "name": "Tom",
4     "github": "https://github.com/tomnomnom/",
5     "likes": ["code", "cheese", "meat"],
6     "contact": {
7         "email": "mail@tomnomnom.com",
8         "twitter": "@TomNomNom"
9     }
10 }
```

```
1 $ fastgron testdata/two.json
2 json = {}
3 json.name = "Tom"
4 json.github = "https://github.com/tomnomnom/"
5 json.likes = []
6 json.likes[0] = "code"
7 json.likes[1] = "cheese"
8 json.likes[2] = "meat"
9 json.contact = {}
10 json.contact.email = "mail@tomnomnom.com"
11 json.contact.twitter = "@TomNomNom"
```

```
1 $ fastgron --help
2 Usage: fastgron [OPTIONS] [FILE | URL] [.path]
3
4 positional arguments:
5   FILE                file name (or '-' for standard input)
6
7 options:
8   -h, --help          show this help message and exit
9   -V, --version       show version information and exit
10  -s, --stream         enable stream mode
11  -F, --fixed-string PATTERN filter output by fixed string.
12                        If -F is provided multiple times, multiple
13                        patterns are searched.
14  -v, --invert-match   select non-matching lines for fixed string search
15  -i, --ignore-case    ignore case distinctions in PATTERN
16  --sort               sort output by key
17  --user-agent         set user agent
18  --header Name:value  set custom HTTP header, can be used multiple
19                        times
20  -u, --ungron         ungron: convert gron output back to JSON
21  -p, -path            filter path, for example .#.3.population or cities.#.
22                        population
23
24                        -p is optional if path starts with . and file with
25                        that name doesn't exist
26
27                        More complex path expressions: .{id,users[1:-3:2]}.{
```

```
22         name,address}}
23         [[3]] is an index accessor without outputting on the
           path.
24         {globalid:id,user:users:[[1]],...} -- path renaming
           with accessor. It's a minimal, limited
           implementation right now.
24     --no-indent    don't indent output
25     --root         root path, default is json
26     --semicolon    add semicolon to the end of each line
27     --no-spaces    don't add spaces around =
28     -c, --color    colorize output
29     --no-color     don't colorize output
```

The file name can be - or missing, in that case the data is read from stdin.

JSON lines (-s or -stream)

```
1  json = []
2  json[0] = {}
3  json[0].one = 1
4  json[0].two = 2
5  json[0].three = []
6  json[0].three[0] = 1
7  json[0].three[1] = 2
8  json[0].three[2] = 3
9  json[1] = {}
10 json[1].one = 1
11 json[1].two = 2
12 json[1].three = []
13 json[1].three[0] = 1
14 json[1].three[1] = 2
15 json[1].three[2] = 3
```

Speed (50x speedup compared to gron on 190MB file)

While there's a 50x speedup for converting JSON to GRON, gron is not able to convert a 800MB file back to JSON.

It takes 8s for fastgron to convert the 840MB file back to JSON.

citylots.json can be downloaded here: <https://github.com/zemirco/sf-city-lots-json/blob/master/citylots.json>

```
1  time fastgron ~/Downloads/citylots.json > /dev/null
2  fastgron ~/Downloads/citylots.json > /dev/null 0.38s user 0.07s system
   99% cpu 0.447 total
3
4  time gron --no-sort ~/Downloads/citylots.json >/dev/null
```

```

5  gron --no-sort ~/Downloads/citylots.json > /dev/null 27.60s user 30.73
   s system 158% cpu 36.705 total
6
7  time fastgron --sort ~/Downloads/citylots.json > /dev/null
8  fastgron --sort ~/Downloads/citylots.json > /dev/null 1.05s user 0.41s
   system 99% cpu 1.464 total
9
10 time gron ~/Downloads/citylots.json > /dev/null
11 gron ~/Downloads/citylots.json > /dev/null 52.34s user 48.46s system
   117% cpu 1:25.80 total
12
13 time fastgron ~/Downloads/citylots.json | rg UTAH
14 json.features[132396].properties.STREET = "UTAH"
15 json.features[132434].properties.STREET = "UTAH"
16 json.features[132438].properties.STREET = "UTAH"
17 json.features[132480].properties.STREET = "UTAH"
18 ...
19 json.features[139041].properties.STREET = "UTAH"
20 json.features[139489].properties.STREET = "UTAH"
21 fastgron ~/Downloads/citylots.json 0.39s user 0.11s system 80% cpu
   0.629 total
22 rg UTAH 0.07s user 0.05s system 19% cpu 0.629 total
23
24 time fastgron -u citylots.gson > c2.json
25 fastgron -u citylots.gson > c2.json 5.62s user 0.47s system 99% cpu
   6.122 total
26
27 time gron -u citylots.gson > c3.json
28 [2] 8270 killed gron -u citylots.gson > c3.json
29 gron -u citylots.gson > c3.json 66.99s user 61.06s system 189% cpu
   1:07.75 total

```

Path finding is 18x faster than jq and 5x faster than jj:

- jq: 3.252s

```

1  $ time jq -cM ".features[10000].properties.LOT_NUM" < ~/Downloads/
   citylots.json
2  "091"
3  jq -cM ".features[10000].properties.LOT_NUM" < ~/Downloads/
   citylots.json 2.91s user 0.28s system 97% cpu 3.252 total

```

- jj: 0.972s console \$ time jj -r features.10000.properties.LOT_NUM < ~/Downloads/citylots.json "091"jj -r features.10000.properties.LOT_NUM < ~/Downloads/citylots.json 0.87s user 0.71s system 161% cpu 0.972 total
- fastgron: 0.176s console \$ time build/fastgron .features.10000.properties.LOT_NUM < ~/Downloads/citylots.json json.features

```
[10000].properties.LOT_NUM = "091"build/fastgron .features.10000.  
properties.LOT_NUM < ~/Downloads/citylots.json 0.07s user 0.10s  
system 95% cpu 0.176 total
```

Building

To build and run this project, you need:

- A C++20 compatible compiler
- CMake (version 3.8 or higher)
- libcurl installed (Optional)

Nix

1. Enable Flakes locally.
2. Run `nix build` to build the `fastgron` binary locally.

For a development environment, either:

- Enable `direnv` in your shell then approve the use of `.envrc` via `direnv allow` or
- explicitly enter a development shell via `nix develop`.

Ubuntu

Here are the steps to build, test, and install `fastgron`:

```
1 apt install cmake clang libcurl4-openssl-dev libssl-dev zlib1g-dev  
2 git clone https://github.com/adamritter/fastgron.git  
3 cd fastgron  
4 cmake -B build -DCMAKE_CXX_COMPILER=/usr/bin/clang++ && cmake --build  
   build  
5 cmake install build/
```

Future development ideas:

- Paths: Implement more complex path queries: using `*`, `[]`, multiple exclusive paths using `{}`. `{}` also could be extended for allowing simple path renaming and value setting, like `{name:.author.name,address:.author.address,is_person:true}`
- Path autocompletion is much better with `gron` type paths than `js` style functions, the code should take advantage of it

-
- memory mapping would be great, but it depends on the underlying SIMDJSON library not needing padding.
 - CSV support would probably be helpful (using csv2 header only library for example), as there are some big CSV files out there. toml / yaml support is not out of the question, but I don't know about people using it in general
 - multiple file support would probably be great, which would make it easy to merge multiple files (especially with giving different `-root` values to different files)
 - after the filters get useful enough, directly outputting JSON is also an option, it can be much faster than gron and then ungron together, as there's no need to build up maps of values
 - for streaming / ndjson, multi-threaded implementation should be used
 - the code should be accessible as a library as well, especially when it gets more powerful
 - simply appending GRON code, like setting some paths/values maybe a useful simple feature
 - A fastjq implementation could be created from the learnings of this project
 - `-skip-initializations`, `-output-json`
 - Beat other benchmarks: <https://colab.research.google.com/github/dcmoura/spyql/blob/master/notebooks/js>