tidwall / **gjson** Public

Get JSON values quickly - JSON parser for Go

MIT license

11.7k stars **761** forks

Star Watch

Code Issues 57 Pull requests 3 Actions Security Insights

master

tidwall ... on Nov 22, 2022

View code

GJSON

api reference playground {} syntax

get json values quickly

GJSON is a Go package that provides a fast and simple way to get values from a json document. It has features such as one line retrieval, dot notation paths, iteration, and parsing json lines.

Also check out SJSON for modifying json, and the JJ command line tool.

This README is a quick overview of how to use GJSON, for more information check out GJSON Syntax.

GJSON is also available for Python and Rust

Getting Started

Installing

To start using GJSON, install Go and run `go get`:

```
$ go get -u github.com/tidwall/gjson
```

This will retrieve the library.

Get a value

Get searches json for the specified path. A path is in dot syntax, such as "name.last" or "age". When the value is found it's returned immediately.

```
package main

import "github.com/tidwall/gjson"

const json = `{"name":{"first":"Janet","last":"Prichard"},"age":47}`

func main() {
        value := gjson.Get(json, "name.last")
        println(value.String())
}
```

This will print:

```
Prichard
```

There's also the GetMany function to get multiple values at once, and GetBytes for working with JSON byte slices.

Path Syntax

Below is a quick overview of the path syntax, for more complete information please check out GJSON Syntax.

A path is a series of keys separated by a dot. A key may contain special wildcard characters '*' and '?'. To access an array value use the index as the key. To get the number of elements in an array or to access a child path, use the '#' character. The dot and wildcard characters can be escaped with '\'.

```
{
  "name": {"first": "Tom", "last": "Anderson"},
```

```
  "age":37,
  "children": ["Sara","Alex","Jack"],
  "fav.movie": "Deer Hunter",
  "friends": [
    {"first": "Dale", "last": "Murphy", "age": 44, "nets": ["ig", "fb", "tw"]},
    {"first": "Roger", "last": "Craig", "age": 68, "nets": ["fb", "tw"]},
    {"first": "Jane", "last": "Murphy", "age": 47, "nets": ["ig", "tw"]}
  ]
}
```

```
"name.last"          >> "Anderson"
"age"                >> 37
"children"           >> ["Sara","Alex","Jack"]
"children.#"         >> 3
"children.1"         >> "Alex"
"child*.2"           >> "Jack"
"c?ildren.0"         >> "Sara"
"fav\.movie"         >> "Deer Hunter"
"friends.#.first"    >> ["Dale","Roger","Jane"]
"friends.1.last"     >> "Craig"
```

You can also query an array for the first match by using `#(...)`, or find all matches with `#(...)#`. Queries support the `==`, `!=`, `<`, `<=`, `>`, `>=` comparison operators and the simple pattern matching `%` (like) and `!%` (not like) operators.

```
friends.#(last=="Murphy").first    >> "Dale"
friends.#(last=="Murphy")#.first   >> ["Dale","Jane"]
friends.#(age>45)#.last            >> ["Craig","Murphy"]
friends.#(first%"D*").last         >> "Murphy"
friends.#(first!%"D*").last        >> "Craig"
friends.#(nets.#(=="fb"))#.first   >> ["Dale","Roger"]
```

Please note that prior to v1.3.0, queries used the `#[...]` brackets. This was changed in v1.3.0 as to avoid confusion with the new multipath syntax. For backwards compatibility, `#[...]` will continue to work until the next major release.

Result Type

GJSON supports the json types `string`, `number`, `bool`, and `null`. Arrays and Objects are returned as their raw json types.

The `Result` type holds one of these:

```
bool, for JSON booleans
float64, for JSON numbers
```

```
string, for JSON string literals
nil, for JSON null
```

To directly access the value:

```
result.Type           // can be String, Number, True, False, Null, or JSON
result.Str            // holds the string
result.Num            // holds the float64 number
result.Raw            // holds the raw json
result.Index          // index of raw value in original json, zero means index
result.Indexes        // indexes of all the elements that match on a path cont
```

There are a variety of handy functions that work on a result:

```
result.Exists() bool
result.Value() interface{}
result.Int() int64
result.Uint() uint64
result.Float() float64
result.String() string
result.Bool() bool
result.Time() time.Time
result.Array() []gjson.Result
result.Map() map[string]gjson.Result
result.Get(path string) Result
result.ForEach(iterator func(key, value Result) bool)
result.Less(token Result, caseSensitive bool) bool
```

The `result.Value()` function returns an `interface{}` which requires type assertion and is one of the following Go types:

```
boolean >> bool
number  >> float64
string  >> string
null    >> nil
array   >> []interface{}
object  >> map[string]interface{}
```

The `result.Array()` function returns back an array of values. If the result represents a non-existent value, then an empty array will be returned. If the result is not a JSON array, the return value will be an array containing one result.

64-bit integers

The `result.Int()` and `result.Uint()` calls are capable of reading all 64 bits, allowing for large JSON integers.

```
result.Int() int64    // -9223372036854775808 to 9223372036854775807
result.Uint() uint64   // 0 to 18446744073709551615
```

Modifiers and path chaining

New in version 1.2 is support for modifier functions and path chaining.

A modifier is a path component that performs custom processing on the json.

Multiple paths can be "chained" together using the pipe character. This is useful for getting results from a modified query.

For example, using the built-in `@reverse` modifier on the above json document, we'll get `children` array and reverse the order:

```
"children|@reverse"           >> ["Jack","Alex","Sara"]
"children|@reverse|0"         >> "Jack"
```

There are currently the following built-in modifiers:

- `@reverse` : Reverse an array or the members of an object.
- `@ugly` : Remove all whitespace from a json document.
- `@pretty` : Make the json document more human readable.
- `@this` : Returns the current element. It can be used to retrieve the root element.
- `@valid` : Ensure the json document is valid.
- `@flatten` : Flattens an array.
- `@join` : Joins multiple objects into a single object.
- `@keys` : Returns an array of keys for an object.
- `@values` : Returns an array of values for an object.
- `@tostr` : Converts json to a string. Wraps a json string.
- `@fromstr` : Converts a string from json. Unwraps a json string.
- `@group` : Groups arrays of objects. See e4fc67c.

Modifier arguments

A modifier may accept an optional argument. The argument can be a valid JSON document or just characters.

For example, the `@pretty` modifier takes a json object as its argument.

```
@pretty:{"sortKeys":true}
```

Which makes the json pretty and orders all of its keys.

```
{
  "age":37,
  "children": ["Sara","Alex","Jack"],
  "fav.movie": "Deer Hunter",
  "friends": [
    {"age": 44, "first": "Dale", "last": "Murphy"},
    {"age": 68, "first": "Roger", "last": "Craig"},
    {"age": 47, "first": "Jane", "last": "Murphy"}
  ],
  "name": {"first": "Tom", "last": "Anderson"}
}
```

The full list of `@pretty` options are `sortKeys`, `indent`, `prefix`, and `width`. Please see Pretty Options for more information.

Custom modifiers

You can also add custom modifiers.

For example, here we create a modifier that makes the entire json document upper or lower case.

```
gjson.AddModifier("case", func(json, arg string) string {
  if arg == "upper" {
    return strings.ToUpper(json)
  }
  if arg == "lower" {
    return strings.ToLower(json)
  }
  return json
})
```

```
"children|@case:upper"           >> ["SARA","ALEX","JACK"]
"children|@case:lower|@reverse"  >> ["jack","alex","sara"]
```

JSON Lines

There's support for JSON Lines using the `..` prefix, which treats a multilined document as an array.

For example:

```
{"name": "Gilbert", "age": 61}
{"name": "Alexa", "age": 34}
{"name": "May", "age": 57}
{"name": "Deloise", "age": 44}
```

```
..#                  >> 4
..1                  >> {"name": "Alexa", "age": 34}
..3                  >> {"name": "Deloise", "age": 44}
..#.name             >> ["Gilbert","Alexa","May","Deloise"]
..#(name="May").age  >> 57
```

The `ForEachLines` function will iterate through JSON lines.

```
gjson.ForEachLine(json, func(line gjson.Result) bool{
    println(line.String())
    return true
})
```

Get nested array values

Suppose you want all the last names from the following json:

```
{
  "programmers": [
    {
      "firstName": "Janet",
      "lastName": "McLaughlin",
    }, {
      "firstName": "Elliotte",
      "lastName": "Hunter",
    }, {
      "firstName": "Jason",
      "lastName": "Harold",
    }
  ]
}
```

You would use the path "programmers.#.lastName" like such:

```
result := gjson.Get(json, "programmers.#.lastName")
for _, name := range result.Array() {
```

```
        println(name.String())
}
```

You can also query an object inside an array:

```
name := gjson.Get(json, `programmers.#(lastName="Hunter").firstName`)
println(name.String())  // prints "Elliotte"
```

Iterate through an object or array

The `ForEach` function allows for quickly iterating through an object or array. The key and value are passed to the iterator function for objects. Only the value is passed for arrays. Returning `false` from an iterator will stop iteration.

```
result := gjson.Get(json, "programmers")
result.ForEach(func(key, value gjson.Result) bool {
        println(value.String())
        return true // keep iterating
})
```

Simple Parse and Get

There's a `Parse(json)` function that will do a simple parse, and `result.Get(path)` that will search a result.

For example, all of these will return the same result:

```
gjson.Parse(json).Get("name").Get("last")
gjson.Get(json, "name").Get("last")
gjson.Get(json, "name.last")
```

Check for the existence of a value

Sometimes you just want to know if a value exists.

```
value := gjson.Get(json, "name.last")
if !value.Exists() {
        println("no last name")
} else {
        println(value.String())
}

// Or as one step
```

```
if gjson.Get(json, "name.last").Exists() {
        println("has a last name")
}
```

Validate JSON

The `Get*` and `Parse*` functions expects that the json is well-formed. Bad json will not panic, but it may return back unexpected results.

If you are consuming JSON from an unpredictable source then you may want to validate prior to using GJSON.

```
if !gjson.Valid(json) {
        return errors.New("invalid json")
}
value := gjson.Get(json, "name.last")
```

Unmarshal to a map

To unmarshal to a `map[string]interface{}`:

```
m, ok := gjson.Parse(json).Value().(map[string]interface{})
if !ok {
        // not a map
}
```

Working with Bytes

If your JSON is contained in a `[]byte` slice, there's the GetBytes function. This is preferred over `Get(string(data), path)`.

```
var json []byte = ...
result := gjson.GetBytes(json, path)
```

If you are using the `gjson.GetBytes(json, path)` function and you want to avoid converting `result.Raw` to a `[]byte`, then you can use this pattern:

```
var json []byte = ...
result := gjson.GetBytes(json, path)
var raw []byte
if result.Index > 0 {
    raw = json[result.Index:result.Index+len(result.Raw)]
```

```
} else {
    raw = []byte(result.Raw)
}
```

This is a best-effort no allocation sub slice of the original json. This method utilizes the `result.Index` field, which is the position of the raw data in the original json. It's possible that the value of `result.Index` equals zero, in which case the `result.Raw` is converted to a `[]byte`.

Get multiple values at once

The `GetMany` function can be used to get multiple values at the same time.

README.md

The return value is a `[]Result`, which will always contain exactly the same number of items as the input paths.

Performance

Benchmarks of GJSON alongside encoding/json, ffjson, EasyJSON, jsonparser, and json-iterator

```
BenchmarkGJSONGet-16                11644512       311 ns/op       0 B/op
0 allocs/op
BenchmarkGJSONUnmarshalMap-16        1122678      3094 ns/op    1920 B/op
26 allocs/op
BenchmarkJSONUnmarshalMap-16          516681      6810 ns/op    2944 B/op
69 allocs/op
BenchmarkJSONUnmarshalStruct-16       697053      5400 ns/op     928 B/op
13 allocs/op
BenchmarkJSONDecoder-16               330450     10217 ns/op    3845 B/op
160 allocs/op
BenchmarkFFJSONLexer-16              1424979      2585 ns/op     880 B/op
8 allocs/op
BenchmarkEasyJSONLexer-16            3000000       729 ns/op     501 B/op
5 allocs/op
BenchmarkJSONParserGet-16            3000000       366 ns/op      21 B/op
0 allocs/op
BenchmarkJSONIterator-16             3000000       869 ns/op     693 B/op
14 allocs/op
```

JSON document used:

```
{
  "widget": {
    "debug": "on",
    "window": {
      "title": "Sample Konfabulator Widget",
      "name": "main_window",
      "width": 500,
      "height": 500
    },
    "image": {
      "src": "Images/Sun.png",
      "hOffset": 250,
      "vOffset": 250,
      "alignment": "center"
    },
    "text": {
      "data": "Click Here",
      "size": 36,
      "style": "bold",
      "vOffset": 100,
      "alignment": "center",
      "onMouseUp": "sun1.opacity = (sun1.opacity / 100) * 90;"
    }
  }
}
```

Each operation was rotated through one of the following search paths:

```
widget.window.name
widget.image.hOffset
widget.text.onMouseUp
```

These benchmarks were run on a MacBook Pro 16" 2.4 GHz Intel Core i9 using Go 1.17 and can be found here.

Releases

60 tags

Packages

No packages published

Used by 13.7k

+ 13,682

Contributors 26

+ 15 contributors

Languages

- **Go** 100.0%