


 mailru / **easyjson** Public

Fast JSON serializer for golang.

 MIT license 4k stars  392 forks Star Watch ▾[Code](#) [Issues](#) 58 [Pull requests](#) 10 [Actions](#) [Projects](#) [Security](#) [Insights](#) master ▾

...



erikdubbelboer ...

✗ on Apr 4, 2022

[View code](#)**easyjson**    A+

Package easyjson provides a fast and easy way to marshal/unmarshal Go structs to/from JSON without the use of reflection. In performance tests, easyjson outperforms the standard `encoding/json` package by a factor of 4-5x, and other JSON encoding packages by a factor of 2-3x.

easyjson aims to keep generated Go code simple enough so that it can be easily optimized or fixed. Another goal is to provide users with the ability to customize the generated code by providing options not available with the standard `encoding/json` package, such as generating "snake_case" names or enabling `omitempty` behavior by default.

Usage

Install:

```
# for Go < 1.17
go get -u github.com/mailru/easyjson/...
```

or

```
# for Go >= 1.17
go get github.com/mailru/easyjson && go install github.com/mailru/easyjson/...@latest
```

Run:

```
easyjson -all <file>.go
```

The above will generate `<file>_easyjson.go` containing the appropriate marshaler and unmarshaler funcs for all structs contained in `<file>.go`.

Please note that easyjson requires a full Go build environment and the `GOPATH` environment variable to be set. This is because easyjson code generation invokes `go run` on a temporary file (an approach to code generation borrowed from [ffjson](#)).

Serialize

```
someStruct := &SomeStruct{Field1: "val1", Field2: "val2"}
rawBytes, err := easyjson.Marshal(someStruct)
```

Deserialize

```
someStruct := &SomeStruct{}
err := easyjson.Unmarshal(rawBytes, someStruct)
```

Please see the [GoDoc](#) for more information and features.

Options

Usage of easyjson:

```
-all
    generate marshaler/unmarshalers for all structs in a file
-build_tags string
    build tags to add to generated file
-gen_build_flags string
    build flags when running the generator while bootstrapping
-byte
    use simple bytes instead of Base64Bytes for slice of bytes
-leave_temps
    do not delete temporary files
-no_std_marshalers
    don't generate MarshalJSON/UnmarshalJSON funcs
-noformat
    do not run 'gofmt -w' on output file
-omit_empty
    omit empty fields by default
-output_filename string
    specify the filename of the output
-pkg
    process the whole package instead of just the given file
-snake_case
    use snake_case names instead of CamelCase by default
-lower_camel_case
    use lowerCamelCase instead of CamelCase by default
-stubs
    only generate stubs for marshaler/unmarshaler funcs
```

```
-disallow_unknown_fields
    return error if some unknown field in json appeared
-disable_members_unescape
    disable unescaping of \uXXXX string sequences in member names
```

Using `-all` will generate marshalers/unmarshalers for all Go structs in the file excluding those structs whose preceding comment starts with `easyjson:skip`. For example:

```
//easyjson:skip
type A struct {}
```

If `-all` is not provided, then only those structs whose preceding comment starts with `easyjson:json` will have marshalers/unmarshalers generated. For example:

```
//easyjson:json
type A struct {}
```

Additional option notes:

`-snake_case` tells easyjson to generate `snake_case` field names by default (unless overridden by a field tag). The CamelCase to `snake_case` conversion algorithm should work in most cases (ie, `HTTPVersion` will be converted to `"http_version"`).

`-build_tags` will add the specified build tags to generated Go sources.

`-gen_build_flags` will execute the easyjson bootstrapping code to launch the actual generator command with provided flags. Multiple arguments should be separated by space e.g. `-gen_build_flags="-mod=mod -x"`.

Structure json tag options

Besides standard json tag options like 'omitempty' the following are supported:

'nocopy' - disables allocation and copying of string values, making them refer to original json buffer memory. This works great for short lived objects which are not hold in memory after decoding and immediate usage. Note if string requires unescaping it will be processed as normally.

'intern' - string "interning" (deduplication) to save memory when the very same string dictionary values are often met all over the structure. See below for more details.

Generated Marshaler/Unmarshaler Funcs

For Go struct types, easyjson generates the funcs `MarshalEasyJSON` / `UnmarshalEasyJSON` for marshaling/unmarshaling JSON. In turn, these satisfy the `easyjson.Marshaler` and `easyjson.Unmarshaler` interfaces and when used in conjunction with `easyjson.Marshal` / `easyjson.Unmarshal` avoid unnecessary reflection / type assertions during marshaling/unmarshaling to/from JSON for Go structs.

easyjson also generates `MarshalJSON` and `UnmarshalJSON` funcs for Go struct types compatible with the standard `json.Marshaler` and `json.Unmarshaler` interfaces. Please be aware that using the standard `json.Marshal` / `json.Unmarshal` for marshaling/unmarshaling will incur a significant performance penalty when compared to using `easyjson.Marshal` / `easyjson.Unmarshal`.

Additionally, easyjson exposes utility funcs that use the `MarshalEasyJSON` and `UnmarshalEasyJSON` for marshaling/unmarshaling to and from standard readers and writers. For example, easyjson provides `easyjson.MarshalToHTTPResponseWriter` which marshals to the standard `http.ResponseWriter`. Please see the [GoDoc listing](#) for the full listing of utility funcs that are available.

Controlling easyjson Marshaling and Unmarshaling Behavior

Go types can provide their own `MarshalEasyJSON` and `UnmarshalEasyJSON` funcs that satisfy the `easyjson.Marshaler` / `easyjson.Unmarshaler` interfaces. These will be used by `easyjson.Marshal` and `easyjson.Unmarshal` when defined for a Go type.

Go types can also satisfy the `easyjson.Optional` interface, which allows the type to define its own `omitempty` logic.

Type Wrappers

easyjson provides additional type wrappers defined in the `easyjson/opt` package. These wrap the standard Go primitives and in turn satisfy the easyjson interfaces.

The `easyjson/opt` type wrappers are useful when needing to distinguish between a missing value and/or when needing to specifying a default value. Type wrappers allow easyjson to avoid additional pointers and heap allocations and can significantly increase performance when used properly.

Memory Pooling

easyjson uses a buffer pool that allocates data in increasing chunks from 128 to 32768 bytes. Chunks of 512 bytes and larger will be reused with the help of `sync.Pool`. The maximum size of a chunk is bounded to reduce redundant memory allocation and to allow larger reusable buffers.

easyjson's custom allocation buffer pool is defined in the `easyjson/buffer` package, and the default behavior pool behavior can be modified (if necessary) through a call to `buffer.Init()` prior to any marshaling or unmarshaling. Please see the [GoDoc listing](#) for more information.

String interning

During unmarshaling, `string` field values can be optionally [interned](#) to reduce memory allocations and usage by deduplicating strings in memory, at the expense of slightly increased CPU usage.

This will work effectively only for `string` fields being decoded that have frequently the same value (e.g. if you have a string field that can only assume a small number of possible values).

To enable string interning, add the `intern` keyword tag to your `json` tag on `string` fields, e.g.:

```
type Foo struct {
    UUID string `json:"uuid"` // will not be interned during unmarshaling
    State string `json:"state,intern"` // will be interned during unmarshaling
}
```

Issues, Notes, and Limitations

☰ README.md

easyjson is still early in its development. As such, there are likely to be bugs and missing features when compared to `encoding/json`. In the case of a missing feature or bug, please create a GitHub issue. Pull requests are welcome!

Unlike `encoding/json`, object keys are case-sensitive. Case-insensitive matching is not currently provided due to the significant performance hit when doing case-insensitive key matching. In the future, case-insensitive object key matching may be provided via an option to the generator.

easyjson makes use of `unsafe`, which simplifies the code and provides significant performance benefits by allowing no-copy conversion from `[]byte` to `string`. That said, `unsafe` is used only when unmarshaling and parsing JSON, and any `unsafe` operations / memory allocations done will be safely deallocated by easyjson. Set the build tag `easyjson_nounsafe` to compile it without `unsafe`.

easyjson is compatible with Google App Engine. The `appengine` build tag (set by App Engine's environment) will automatically disable the use of `unsafe`, which is not allowed in App Engine's Standard Environment. Note that the use with App Engine is still experimental.

Floats are formatted using the default precision from Go's `strconv` package. As such, easyjson will not correctly handle high precision floats when marshaling/unmarshaling JSON. Note, however, that there are very few/limited uses where this behavior is not sufficient for general use. That said, a different package may be needed if precise marshaling/unmarshaling of high precision floats to/from JSON is required.

While unmarshaling, the JSON parser does the minimal amount of work needed to skip over unmatching parens, and as such full validation is not done for the entire JSON value being unmarshaled/parsed.

Currently there is no true streaming support for encoding/decoding as typically for many uses/protocols the final, marshaled length of the JSON needs to be known prior to sending the data. Currently this is not possible with easyjson's architecture.

easyjson parser and codegen based on reflection, so it won't work on `package main` files, because they can't be imported by parser.

Benchmarks

Most benchmarks were done using the example [13kB example JSON](#) (9k after eliminating whitespace). This example is similar to real-world data, is well-structured, and contains a healthy variety of different types, making it ideal for JSON serialization benchmarks.

Note:

For small request benchmarks, an 80 byte portion of the above example was used.

For large request marshaling benchmarks, a struct containing 50 regular samples was used, making a ~500kB output JSON.

Benchmarks are showing the results of easyjson's default behaviour, which makes use of `unsafe`.

Benchmarks are available in the repository and can be run by invoking `make`.

easyjson vs. encoding/json

easyjson is roughly 5-6 times faster than the standard `encoding/json` for unmarshaling, and 3-4 times faster for non-concurrent marshaling. Concurrent marshaling is 6-7x faster if marshaling to a writer.

easyjson vs. ffjson

easyjson uses the same approach for JSON marshaling as [ffjson](#), but takes a significantly different approach to lexing and parsing JSON during unmarshaling. This means easyjson is roughly 2-3x faster for unmarshaling and 1.5-2x faster for non-concurrent unmarshaling.

As of this writing, `ffjson` seems to have issues when used concurrently: specifically, large request pooling hurts `ffjson`'s performance and causes scalability issues. These issues with `ffjson` can likely be fixed, but as of writing remain outstanding/known issues with `ffjson`.

easyjson and `ffjson` have similar performance for small requests, however easyjson outperforms `ffjson` by roughly 2-5x times for large requests when used with a writer.

easyjson vs. go/codec

[go/codec](#) provides compile-time helpers for JSON generation. In this case, helpers do not work like marshalers as they are encoding-independent.

easyjson is generally 2x faster than `go/codec` for non-concurrent benchmarks and about 3x faster for concurrent encoding (without marshaling to a writer).

In an attempt to measure marshaling performance of `go/codec` (as opposed to allocations/memcpy/writer interface invocations), a benchmark was done with resetting length of a byte slice rather than resetting the whole slice to nil. However, the optimization in this exact form may not be applicable in practice, since the memory is not freed between marshaling operations.

easyjson vs 'ujson' python module

[ujson](#) is using C code for parsing, so it is interesting to see how plain golang compares to that. It is important to note that the resulting object for python is slower to access, since the library parses JSON object into dictionaries.

easyjson is slightly faster for unmarshaling and 2-3x faster than `ujson` for marshaling.

Benchmark Results

ffjson results are from February 4th, 2016, using the latest ffjson and go1.6. go/codec results are from March 4th, 2016, using the latest go/codec and go1.6.

Unmarshaling

lib	json size	MB/s	allocs/op	B/op
standard	regular	22	218	10229
standard	small	9.7	14	720
easyjson	regular	125	128	9794
easyjson	small	67	3	128
ffjson	regular	66	141	9985
ffjson	small	17.6	10	488
codec	regular	55	434	19299
codec	small	29	7	336
ujson	regular	103	N/A	N/A

Marshaling, one goroutine.

lib	json size	MB/s	allocs/op	B/op
standard	regular	75	9	23256
standard	small	32	3	328
standard	large	80	17	1.2M
easyjson	regular	213	9	10260
easyjson*	regular	263	8	742
easyjson	small	125	1	128
easyjson	large	212	33	490k
easyjson*	large	262	25	2879
ffjson	regular	122	153	21340
ffjson**	regular	146	152	4897
ffjson	small	36	5	384
ffjson**	small	64	4	128
ffjson	large	134	7317	818k

lib	json size	MB/s	allocs/op	B/op
ffjson**	large	125	7320	827k
codec	regular	80	17	33601
codec***	regular	108	9	1153
codec	small	42	3	304
codec***	small	56	1	48
codec	large	73	483	2.5M
codec***	large	103	451	66007
ujson	regular	92	N/A	N/A

* marshaling to a writer, ** using `ffjson.Pool()`, *** reusing output slice instead of resetting it to nil


Marshaling, concurrent.

lib	json size	MB/s	allocs/op	B/op
standard	regular	252	9	23257
standard	small	124	3	328
standard	large	289	17	1.2M
easyjson	regular	792	9	10597
easyjson*	regular	1748	8	779
easyjson	small	333	1	128
easyjson	large	718	36	548k
easyjson*	large	2134	25	4957
ffjson	regular	301	153	21629
ffjson**	regular	707	152	5148
ffjson	small	62	5	384
ffjson**	small	282	4	128
ffjson	large	438	7330	1.0M
ffjson**	large	131	7319	820k
codec	regular	183	17	33603
codec***	regular	671	9	1157

lib	json size	MB/s	allocs/op	B/op
codec	small	147	3	304
codec***	small	299	1	48
codec	large	190	483	2.5M
codec***	large	752	451	77574

* marshaling to a writer, ** using `ffjson.Pool()` , *** reusing output slice instead of resetting it to nil

Releases 6

 **Version 0.7.7** Latest


on Feb 6, 2021

[+ 5 releases](#)

Packages

No packages published

Used by 51.7k



+ 51,691

Contributors 80



[+ 69 contributors](#)

Languages

