Jacob Kim Follow

Apr 4 · 7 min read ·

To Unmarshal() or To Decode()? JSON Processing in Go Explained

If you are like me and want to learn backend development, you probably came across JSON handling at one point. JSON is a very popular format for transferring data between the frontend and the backend. Because it is such an important feature in modern web development, Go adds support for JSON in its `encoding/json` package.

The problem is that there isn't only one way of doing things. If you have watched a couple of tutorials in the past, you may have noticed people using different functions to handle JSON. Some people use `Marshal` and `Unmarshal`, while others use `Encode` and `Decode`. What should you use? Which one is better? In this blog post, I will try to explain the difference between the two approaches, and when you should use one over the other. Enjoy!

But first, let me show you the two approaches.

There are two ways to read and write JSON. This code snippet will help you understand how to use the two approaches. First, `Marshal` and `Unmarshal`:

```
func PrettyPrint(v interface{}) (err error) {
    b, err := json.MarshalIndent(v, "", "\t")
    if err == nil {
        fmt.Println(string(b))
    }
    return err
}

func TryMarshal() error {
    data := map[string]interface{}{
        "1": "one",
        "2": "two",
        "3": "three",
    }
    result, err := json.Marshal(&data)
    if err != nil {
        return err
    }

    err = PrettyPrint(result)
    if err != nil {
        return err
    }

    return nil
}

func TryUnmarshal() error {
    myFile, err := os.Open("test.csv")
    if err != nil {
        return err
    }
    defer myFile.Close()

    data, err := io.ReadAll(myFile)
    if err != nil {
        return err
    }

    var result map[string]interface{}
    json.Unmarshal([]byte(data), &result)

    err = PrettyPrint(result)
    if err != nil {
        return err
    }

    return nil
}
```

In `TryMarshal`, I created a `map[string]interface{}` to hold some data. I then passed it to `Marshal`.

In `TryUnmarshal`, I read a file and converted it into byte slice `data`. That `data` is passed into `Unmarshal`, which stores the result in a `map[string]interface{}`.

`PrettyPrint` just formats the output to make it look nice.

Now let's take a look at `Encoder.Encode` and `Decoder.Decode`.

```go
func TryEncode() error {
    data := map[string]interface{}{
        "1": "one",
        "2": "two",
        "3": "three",
    }
    err := json.NewEncoder(os.Stdout).Encode(&data)
    if err != nil {
        return err
    }

    return nil
}

func TryDecode(path string) error {
    myFile, err := os.Open(path)
    if err != nil {
        return err
    }
    defer myFile.Close()

    var result map[string]interface{}
    json.NewDecoder(myFile).Decode(&result)

    return nil
}
```

The code looks pretty similar to the previous example. `TryEncode` is analogous to `TryMarhsal` and `TryDecode` is analogous to `TryUnmarshal`. The only difference here is that `Encode` and `Decode` are methods to `Encoder` and `Decoder` types.

`NewEncoder` takes in the `io.Writer` interface and returns a new `Encoder` type. `NewDecoder` takes in the `io.Reader` interface and returns a new `Decoder` type. For this example, I passed in `os.Stdout` for `NewEncoder` and `myFile` (which is of `os.File` type) for `NewDecoder`.

Now that we know how to use these functions, we can dive into how the two approaches differ under the hood.

Marshal() and Unmarshal()

Let's take a look at their implementation:

```
func Marshal(v any) ([]byte, error) {
    e := newEncodeState()

    err := e.marshal(v, encOpts{escapeHTML: true})
    if err != nil {
        return nil, err
    }
    buf := append([]byte(nil), e.Bytes()...)

    encodeStatePool.Put(e)

    return buf, nil
}

func Unmarshal(data []byte, v any) error {
    var d decodeState
    err := checkValid(data, &d.scan)
    if err != nil {
        return err
    }

    d.init(data)
    return d.unmarshal(v)
}
```

All you need to know here is that:

- `Marshal` takes any value (`any` is a wrapper around `interface{}`) and converts it into a byte slice.
- `Unmarshal` takes a byte slice, parses it, and stores the result to `v`.

Also, take a look at how `Marshal` **stores all the bytes** into a byte slice `buf`. This means that `Marshal` **needs to hold all the data in memory** for it to work. This can be rather RAM-intensive. `Unmarshal` has similar issues because it takes in an entire byte slice as input.

NewEncoder().Encode() and NewDecoder().Decode()

Here's the code for Encode() and Decode():

```go
func (enc *Encoder) Encode(v any) error {
    if enc.err != nil {
        return enc.err
    }
    e := newEncodeState()
    err := e.marshal(v, encOpts{escapeHTML: enc.escapeHTML})
    if err != nil {
        return err
    }

    e.WriteByte('\n')

    b := e.Bytes()
    if enc.indentPrefix != "" || enc.indentValue != "" {
        if enc.indentBuf == nil {
            enc.indentBuf = new(bytes.Buffer)
        }
        enc.indentBuf.Reset()
        err = Indent(enc.indentBuf, b, enc.indentPrefix,
enc.indentValue)
        if err != nil {
            return err
        }
        b = enc.indentBuf.Bytes()
    }
    if _, err = enc.w.Write(b); err != nil {
        enc.err = err
    }
    encodeStatePool.Put(e)
    return err
}

func (dec *Decoder) Decode(v any) error {
    if dec.err != nil {
        return dec.err
    }
    if err := dec.tokenPrepareForDecode(); err != nil {
        return err
    }

    if !dec.tokenValueAllowed() {
        return &SyntaxError{msg: "not at beginning of value", Offset:
dec.InputOffset()}
    }

    n, err := dec.readValue()
    if err != nil {
        return err
    }
    dec.d.init(dec.buf[dec.scanp : dec.scanp+n])
    dec.scanp += n

    err = dec.d.unmarshal(v)

    dec.tokenValueEnd()

    return err
}
```

The code is longer here, but just remember these things:

- `Encode` and `Decode` are methods for `Encoder` and `Decoder` types, which are wrappers around the popular interface `io.Writer` and `io.Reader`.
- `Encode` and `Decode` **streams data instead of storing everything at once.** There is a buffer from which `Encode` and `Decode` writes and read, and this happens until all data is processed.

Ok… so which one should I use?

Good question! I was curious to see the performance differences between the two approaches, so I wrote a test and benchmarked them. Note that any printing has been disabled for the tests.

The test is designed for `Unmarshal` and `Decode` because normally you usually wouldn't write a huge JSON data yourself, while it is possible to receive huge JSON data from a server. You can still expect similar results for `Marshal` and `Encode` because they are basically reverse of their partner functions.

Here is the test code:

```
func BenchmarkTryUnmarshal(b *testing.B) {
    for i := 0; i < b.N; i++ {
        err := TryUnmarshal("file.json")
        if err != nil {
            b.Fatalf("error: %v", err)
        }
    }
}

func BenchmarkTryDecode(b *testing.B) {
    for i := 0; i < b.N; i++ {
        err := TryDecode("file.json")
        if err != nil {
            b.Fatalf("error: %v", err)
        }
    }
}
```

`"file.json"` is our experimental variable. These will be JSON files of different sizes for each run. The first five JSON files are sourced from JSONPlaceholder - Free Fake REST API. The last JSON file (the largest one) is sourced from test-data/large-file.json at master · json-iterator/test-data · GitHub.

Here is the architecture used for the test.

```
goos: linux
goarch: amd64
pkg: example.com/jsonExperiment
cpu: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
```

And here are the benchmark data.

Unmarshal

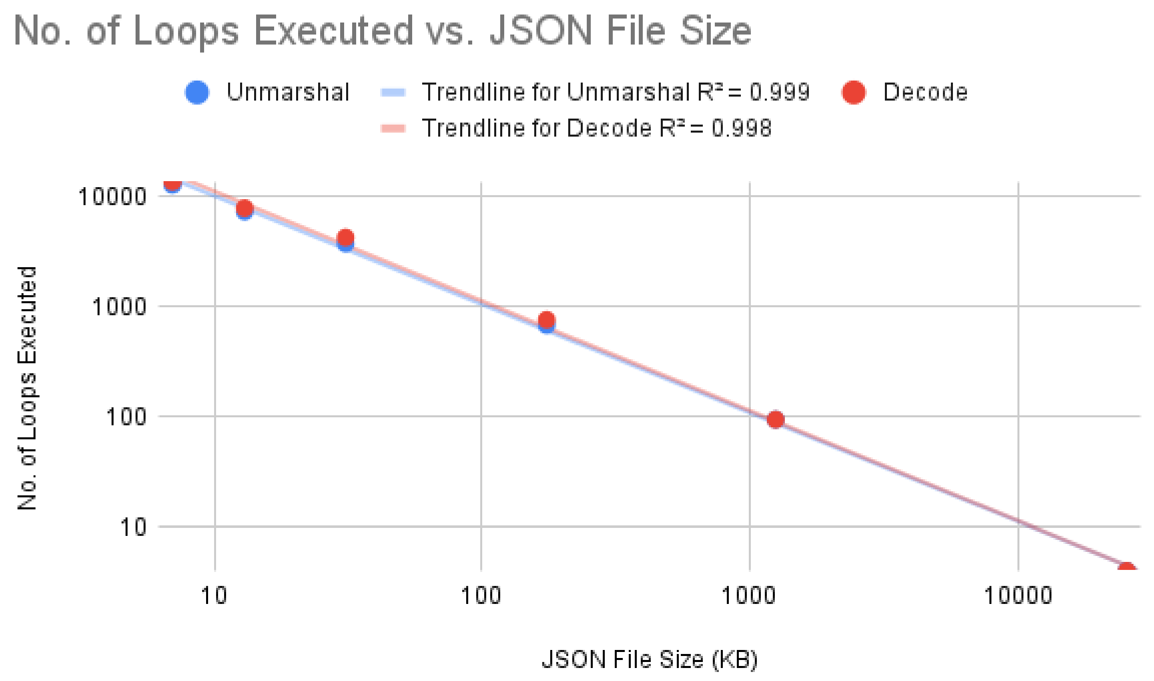
JSON File Size	No. of Loops	Avg. Amount of Time Taken per	Avg. No. of Bytes Allocated per	Avg. No. of Allocations per Operation

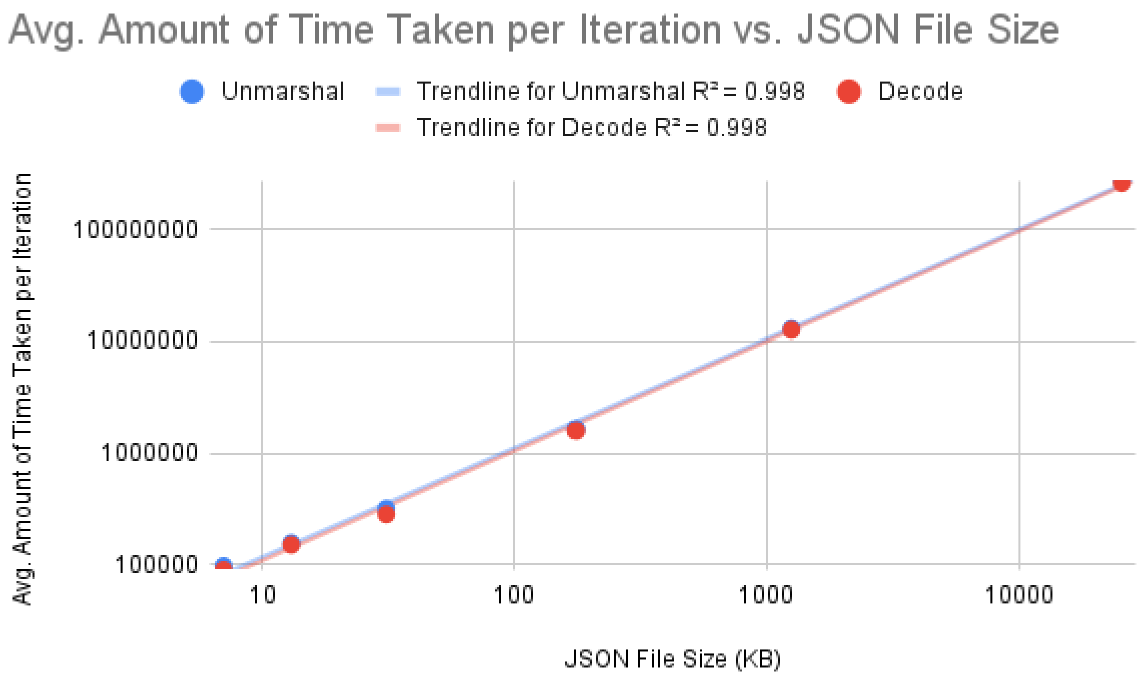
(KB)	Executed	Iteration (ns/op)	Operation (B/op)	(allocs/op)
7	12819	96463	56520	582
13	7323	155131	112736	1036
31	3748	315787	238736	1439
175	679	1647401	1300978	8992
1252	94	12934180	10677126	84603
25618	4	276607700	237844490	1750110

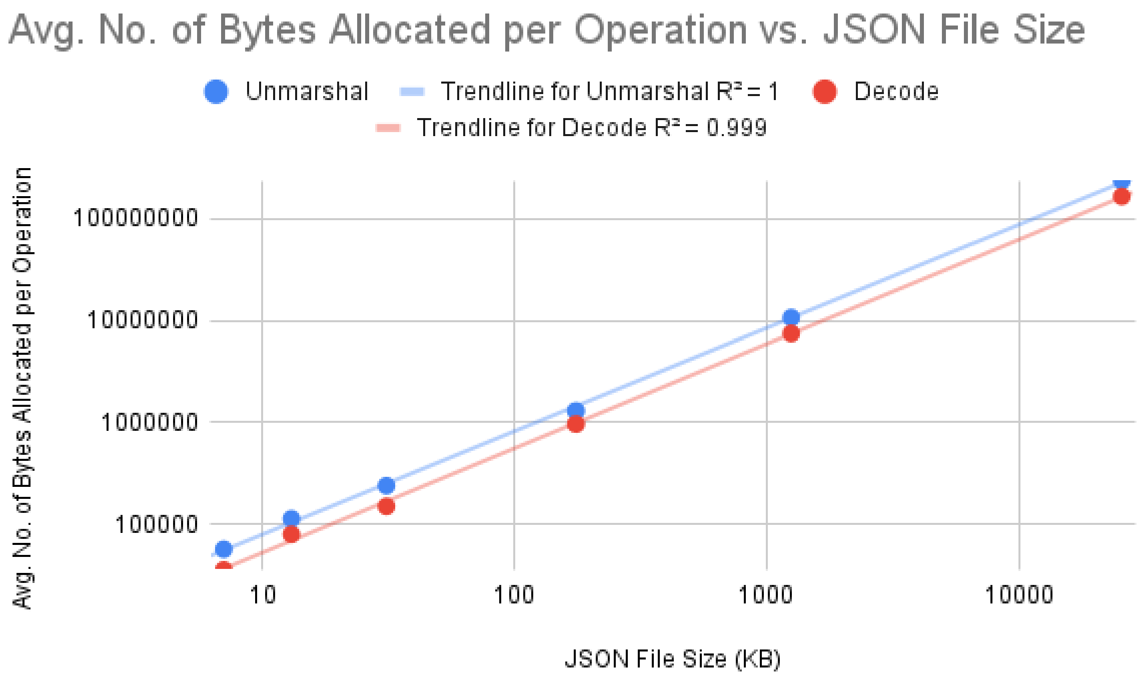
Decode

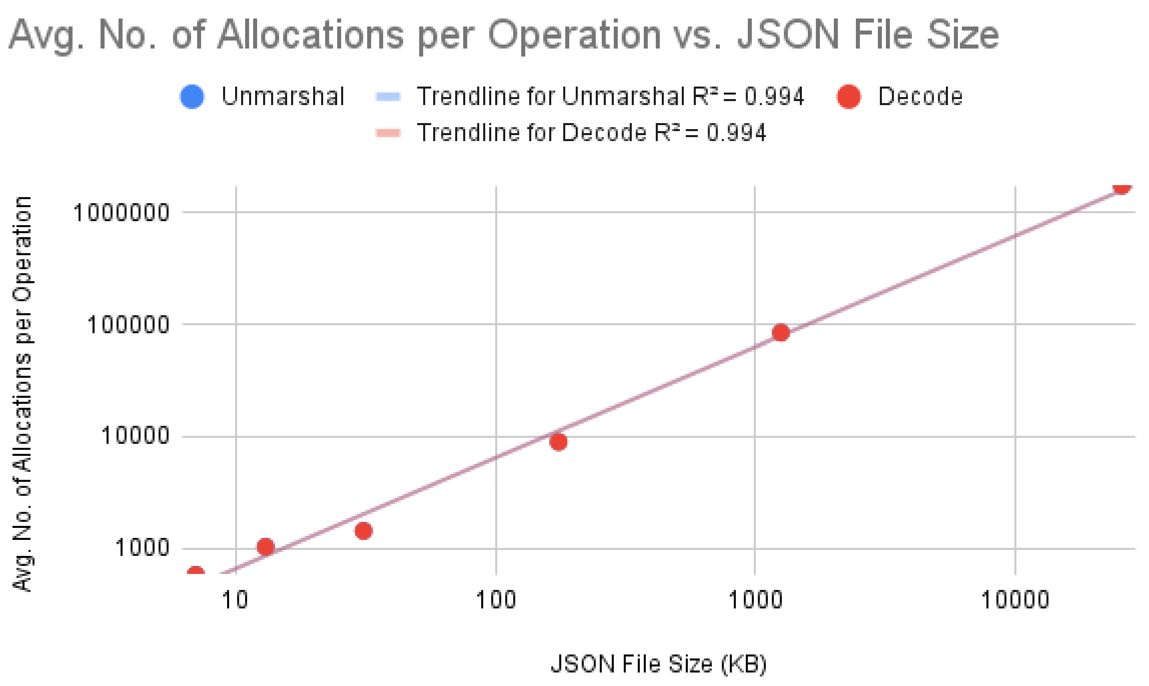
JSON File Size (KB)	No. of Loops Executed	Avg. Amount of Time Taken per Iteration (ns/op)	Avg. No. of Bytes Allocated per Operation (B/op)	Avg. No. of Allocations per Operation (allocs/op)
7	13702	88839	35432	580
13	7882	149191	79312	1032
31	4260	280336	149424	1433
175	760	1583824	965830	8983

1252	94	12613618	7491156	84588
25618	4	261644025	166432316	1750103




Avg. Amount of Time Taken per Iteration vs. JSON File Size
Unmarshal
Trendline for Unmarshal R² = 0.998
Decode
Trendline for Decode R² = 0.998
Avg. Amount of Time Taken per Iteration
100000000
10000000
1000000
100000
10
100
1000
10000
JSON File Size (KB)


Avg. No. of Bytes Allocated per Operation vs. JSON File Size
● Unmarshal
Trendline for Unmarshal R² = 1
● Decode
Trendline for Decode R² = 0.999
Avg. No. of Bytes Allocated per Operation
100000000
10000000
1000000
100000
10
100
1000
10000
JSON File Size (KB)



We can see some patterns here:

- `Decode` consistently uses less memory than `Unmarshal`. However, this isn't a lot of difference.
- Everything else doesn’t differ that much.
- Memory usage seems to become an issue when you are dealing with large JSON files, but it’s unlinkely that a server will receive a JSON data that huge. I was pushing it with the last example.

Conclusion

The difference in performance is rather small, so I don't think performance should be a make-or-break factor when it comes to deciding which approach to take. The more reasonable way to consider is to see what data format you are working with. For example, take a look at this snippet.

```
func Homepage(w http.ResponseWriter, r *http.Request){
    type pageData struct {
        visited time.Time
        message string
    }
    homepageData := pageData{time.Now(), "Welcome!"}
    json.NewEncoder(w).Encode(&homepageData)
}

func main() {
    http.HandleFunc("/", Homepage)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

This is a simple example of what an API looks like in Go. Any requests to the `/` endpoint will trigger this code to run the `Homepage` controller, which creates an instance of `pageData` and encodes it using `NewEncoder(w).Encode(&homepageData)`. `w` implements `io.Writer`, so it's more convenient to use `Encode` which expects to be used on `io.Writer`. You could technically convert the struct into byte slice and then use `Marshal`. But why take an extra step when you don't need to?

The takeaway point is that you shouldn’t be worried about performance too much, until it becomes an issue. Instead, you should just pick a solution that is the easiest to use at the given time. If there is a byte slice to work with, use `Marshal` and `Unmarshal`. If there is an `io.Writer` or an `io.Reader`, use `Encode` and `Decode`.

Thank you for reading! This turned out to be an interesting topic for me, and I wanted to run some experiments. Let me know down in the comments if you like these types of posts! You can read this post on Dev.to and my personal site as well.