Go json.Decoder Considered Harmful

Ahmet Alp Balkan published on 28 April 2016

If you are coding with Go and using `json.Decoder` to deserialize a JSON payload, then you are probably signing up for unexpected outcomes. You should use `json.Unmarshal` instead.

1. `json.Decoder` is designed for JSON streams; not for single JSON objects
2. `json.Decoder` silently ignores certain malformed JSON syntax.
3. `json.Decoder` does not release network connection for reuse (slows down HTTP requests up to ~4x)

None of this should be a surprise for someone reading the godoc of the package, but it is. I have done this mistake myself many times. Most developers are finding the method signature of `json.Decoder.Decode(...)` a better fit to parse from an `io.Reader` type than `json.Unmarshal(...)`.

1. json.Decoder is for JSON streams

JSON streams are just concatenated (or newline-separated) JSON values. Here is an example stream:

```
{"Name": "Ed"}{"Name": "Sam"}{"Name": "Bob"}
```

The entire content of this stream is not valid JSON: It should be inside a `[ ]` to be a valid JSON value.

This is just JSON objects concatenated, in other words, it is a valid JSON stream.

The `json.Decoder` type is designed explicitly for JSON streams, full stop. Most probably, **your JSON payload is not this**.

So why the hell JSON streams even exist? Can't we just do JSON arrays? JSON streams are useful for:

- storing structured data in a file and appending easily without parsing the entire file
- streaming structured data live from an API etc (e.g. `docker logs`/`docker events` APIs use this)

If you are dealing with deserialization of a **single** object from JSON, stop using it.

2. json.Decoder silently ignores invalid syntax

Not all invalid syntax, but some strings that are invalid JSON but valid JSON streams can be ignored by `json.Decoder`. Here is an example to illustrate this. Let's say your API is supposed to return:

```
{"Name": "Bob"}
```

but the service introduces some bug and suddenly starts responding with

```
{}{"Name": "Bob"}
```

This is obviously an invalid JSON payload, but it is a valid JSON stream and `json.Decoder` respects that.

But you did not see this coming and your code is just parsing the response into a single JSON object:

```
type Person struct { Name string }

...
var v Person
if err := dec.Decode(&v); err != nil {
    panic(err)
}
fmt.Println(v.Name)
```

you are going to get empty string in `v.Name`, and no error at all. `json.Decoder` has unmarshaled the first JSON object and the rest is just ignored.

Is this likely to happen? Probably not, but can you be 100% sure? Because when it happens you will not be able to debug it so easily.

3. json.Decoder does not drain HTTP connections properly

This is pointed out recently by Filippo Valsorda and you are affected by this unless you are using Go 1.7 (which is not yet released as of this writing).

If you are making an HTTP request and passing the response body to `json.Decoder#Decode()` (what most people does) then most likely your connections are not drained properly, causing possibly a 4x slow down on the HTTP client.

If the HTTP endpoint is responding with a single JSON object and you are calling `json.Decoder#Decode()` only once, it means you are not getting `io.EOF` returned yet. Therefore you are not terminating `json.Decoder` by seeing that `io.EOF` and the response body remains open and therefore the TCP connection (or another Transport used) cannot be returned to the connection pool even though you are done with it. Read more about it here.

This is now fixed in golang master branch and it most probably will get released in Go 1.7.

In the meanwhile, if your response body is small enough, just read it all into memory with `ioutil.ReadAll` and use `json.Unmarshal`. If you want to keep using `json.Decoder`, you need to drain the unread part of the response body with something like:

```
io.Copy(ioutil.Discard, resp.Body)
```

so if you are using `json.Decoder`, go back to your codebase and replace all `defer resp.Body.Close()` with:

```
defer func() {
    io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close()
}()
```

Conclusion

Do not use `json.Decoder` if you are not dealing with JSON streaming.

Use `json.Unmarshal`:

- if you don't know what Go JSON streams are
- if you are working with a single JSON object at a time
- if there is a remote possibility that the API may return malformed JSON

Now you know the trade-offs and use it at your own discretion.